VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA A REKONSTRUKCE WEBOVÉHO PROVOZU

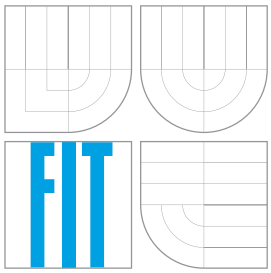
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB OLBERT

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA A REKONSTRUKCE WEBOVÉHO PROVOZU

WEB TRAFFIC ANALYSIS AND RECONSTRUCTION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAKUB OLBERT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VLADIMÍR VESELÝ

BRNO 2012

Abstrakt

Práce popisuje problematiku rekonstrukce a analýzy webového provozu. Hlavním cílem projektu je nastudování teoretické roviny této oblasti a vytvoření návrhu rekonstrukčního nástroje, jehož implementace je předmětem diplomové práce. Nástroj na základě odposlechnutých síťových dat rekonstruuje webový provoz tak, aby mohl být zpětně vizualizován. Využití se předpokládá u bezpečnostních složek státu zabývajících se internetovou kriminalitou.

Abstract

The thesis describes the problems of reconstruction and analysis of web traffic. The main goal of this project is to study theoretical background and to create a design of the reconstruction tool. The tool does reconstruction of the web traffic based on captured network data. Output of the reconstruction process is intended for later visualization. Main usage of this tool is expected at the law enforcing agencies dealing with Internet crime.

Klíčová slova

HTTP, rekonstrukce, analýza, bezpečnost, web

Keywords

HTTP, reconstruction, analysis, security, WWW

Citace

Jakub Olbert: Analýza a rekonstrukce webového provozu, diplomová práce, Brno, FIT VUT v Brně, 2012

Analýza a rekonstrukce webového provozu

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně pod vedením pana inženýra Vladimíra Veselého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Olbert
23. května 2012

Poděkování

Jako poděkování svému vedoucímu panu inženýru Vladimírovi Veselému, který mi věnoval nemálo času při odborných konzultacích diplomové práce, zde uvádím recept na chutné a jednoduché studentské jídlo *Těstoviny s tuňákem a rajčaty.*

Uvaříme těstoviny, přidáme nakrájená rajčata a přimícháme stejk z tuňáka. Zakápneme olivovým olejem, aby pokrm nebyl suchý, a dochutíme Provensálským kořením. Na závěr porci na talíři dozdobíme strouhaným parmezánem. Dobrou chuť!

Obsah

Seznam obrázků

Kapitola 1

Úvod

Diplomová práce si klade za cíl seznámit čtenáře s problematikou rekonstrukce a analýzy webového provozu a představit návrh vlastního rekonstrukčního nástroje, který daný problém řeší prakticky. Hlavním výstupem je pak implementace navrženého nástroje.

Rekonstrukční nástroj je vyvíjen v rámci projektu *Moderní prostředky pro boj s kybernetickou kriminalitou na Internetu nové generace – Sec6net* [6] v rámci skupiny *Reconstruct*, která se zaměřuje na rekonstrukci síťového provozu. Již z názvu projektu je patrná motivace pro řešení problematiky rekonstrukce webového provozu, kterou je bezpečnost počítačových sítí, konkrétně Internetu.

Internetová kriminalita má mnoho podob [18] a mnoho forem páchání. Nemalá část trestných činů je spáchána právě prostřednictvím webových služeb. Cílem této práce je navrhnout a implementovat nástroj provádějící automatizovanou rekonstrukci webového provozu, jehož výstup bude snadno vizualizovatelný. Využití nástroje se předpokládá u bezpečnostních složek státu zabývajících se internetovou kriminalitou.

Kapitola 2 uvede čtenáře do teoretické roviny problematiky rekonstrukce a analýzy webového provozu. Nechybí zde stručný popis protokolu HTTP jakožto hlavního komunikačního protokolu pro webové služby. Druhá část kapitoly je věnována možným řešením rekonstrukce webového provozu.

Následující kapitola 3 popisuje návrh a implementaci prvního z nastíněných řešení, a to *syntaktickou analýzu HTML kódu*. Závěr kapitoly shrnuje výsledky dosažené touto metodou.

Na základě omezení zjištěných u předchozího řešení byla navržena nová architektura rekonstrukčního nástroje, která vychází z použití *HTTP proxy cache*. Nárvh nové architektury včetně jejích vlastností popisuje kapitola 4. Implementaci je věnována kapitola 5. Testování a výkonnostní analýza výsledného nástroje jsou popsány v kapitole 6.

Závěr práce shrnuje dosažené výsledky a diskutuje možnosti využití nástroje.

Kapitola 2

Rekonstrukce webového provozu

Kapitola si klade za cíl uvést čtenáře do problematiky analýzy a rekonstrukce webového provozu v obecné rovině. V úvodní části bude nastíněna motivace pro řešení této problematiky, následovaná krátkým představení protokolu HTTP jakožto hlavního komunikačního prostředku webových aplikací. Druhá polovina kapitoly je zaměřena na vysvětlení základních principů rekonstrukce webového provozu.

2.1 Protokol HTTP

Hypertext Transfer Protocol je aplikační protokol pro distribuované, spolupracující, hypermediální informační systémy [12]. Je to hlavní komunikační prostředek webových aplikací. Jedná se generický textově orientovaný bezstavový protokol. Současná verze má označení 1.1 [12], stále ovšem můžeme narazit na implementace starší verze 1.0 [8]. Verze 1.1 je zpětně kompatibilní s verzí 1.0. V této práci nadále uvažujme verzi 1.1, pokud nebude řečeno jinak.

Komunikační model *klient – server* protokolu HTTP tvoří dvojice zpráv dotaz – odpověď. Různé dvojice mezi sebou nemají žádný vztah, protokol je bezestavový. Webové aplikace vyžadující uchování stavové informace musí řešit tento požadavek ve vlastní režii (např. pomocí metadat *Cookies* přenášených v hlavičkách protokolu HTTP [15]).

Generičnost a široké využití protokolu HTTP v rámci Internetu spočívá v možnostech přenosu libovolného typu dat, díky standardu *MIME* (Multipurpose Internet Mail Extensions, definice v [13] a dalších), a také díky možnosti rozšíření protokolu o další hlavičky bez nutnosti zásahů do stávající implementace webových serverů a bez ztráty kompatibility se standardizovanou verzí protokolu.

Díky jednoduchému komunikačnímu modelu a bezestavosti je usnadněna implementace webových serverů. Nadstandardní služby a aplikace mohou být realizovány jako samostatné softwarové celky využívající webové HTTP servery (např. PHP preprocesor [2] jako rozšíření webového serveru Apache [4]).

2.1.1 Terminologie protokolu HTTP

Následující výčet pojmů seznámí čtenáře se základní terminologií použitou dále v této práci. Význam zde uvedených termínů odpovídá jejich specifikaci uvedené v [12, kap. 1.3]. Některé anglické termíny zůstaly nepřeloženy, neboť nemají vhodný český ekvivalent.

spojení – z pohledu HTTP je chápáno jako spojení na transportní vrstvě mezi dvěma programy, které umožňuje jejich vzájemnou komunikaci.

zpráva – základní jednotka HTTP komunikace, sestávající ze strukturované sekvence oktetů odpovídající syntaxi popsané v kapitole 2.1.3. Existují dva typy zpráv: *dotaz* a *odpověď*. Zprávy jsou přenášeny spojením.

zdroj – datový objekt nebo služba, které lze identifikovat pomocí URI. Zdroj může být dostupný ve více reprezentacích (různé datové formáty, jazykové mutace, ...).

entita – informace přenášená jako užitečná data dotazu nebo odpovědi. Entita sestává z meta-informací ve formě hlaviček zprávy a obsahu, který je uložen v těle zprávy.

klient – je z pohledu HTTP program, který navazuje spojení za účelem zaslání dotazu (2.1.3).

uživatelský agent – je klientem, který iniciuje zaslání dotazu. Nejčastěji se jedná o webový prohlížeč.

server – je aplikační program, který přijímá spojení za účelem zpracování dotazu a zaslání odpovědi. Jako server mohou vystupovat různé uzly: původní server, brána, proxy a tunel.

původní server – server, na kterém je umístěn nebo má být vytvořen zdroj.

proxy – je prostředníkem mezi klientem a serverem, Vystupuje jako klient i jako server. Jedná se o aplikačního agenta, který přijímá požadavky od klientů (zde vystupuje jako server) a přeposílá je dál směrem k původnímu serveru (zde vystupuje jako klient). Analogický postup se opakuje i při přijetí odpovědi od původního serveru, která je přeposlána klientovi.

non-trasparent proxy – provádí modifikace zpráv pouze za účelem autentizace a identifikace proxy (přidává pouze nezbytně nutné autentizační hlavičky).

transparent proxy – může provádět modifikace zpráv za účelem poskytnutí speciálních služeb už. agentům jako například transformace typů médií, anonymní filtrování a redukci protokolu.

gateway (brána) – je server, který je prostředníkem pro jiné další servery. Na rozdíl od proxy přijímá požadavky jako by se jednalo o původní server pro požadovaný zdroj.

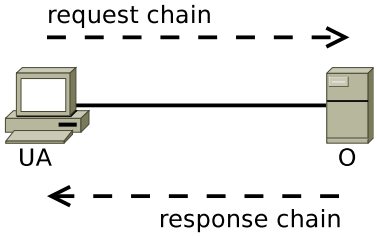
tunnel (tunel) – program vytvářející přemostění mezi dvěma spojeními. Po svém ustavení je tunel transparentní z pohledu HTTP komunikace (není vnímán jako účastník komunikace), ačkoliv jeho vytvoření může být iniciováno HTTP žádostí.

cache – jedná se o programové lokální úložiště (vyrovnávací paměť) HTTP odpovědí. Ukládání, vracení a mazání odpovědí je řízeno kontrolním systémem, který zajišťuje konzistenci a aktuálnost spravovaných dat. Cache ukládá odpovědi serverů, které pak může poskytovat klientům, čímž je snížena zátěž originálních serverů a zkrácena doba zpracování dotazu z pohledu klienta.

2.1.2 Komunikační řetězec protokolu HTTP

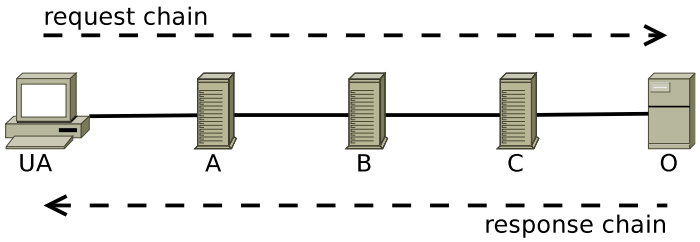
Z pohledu analýzy je rovněž důležité se seznámit s komunikačním řetězcem protokolu HTTP. Komunikace obvykle probíhá nad transportním protokolem TCP a síťovým protokolem IP (případně IPv6), výchozí port číslo 80.

HTTP komunikace je nejčastěji iniciována uživatelským klientem, který zašle dotaz původnímu serveru. Nejjednodušším případem komunikačního řetězce je přímé spojení mezi uživatelským agentem (UA) a původním serverem (O), které je znázorněné obrázkem 2.1.



Obrázek 2.1: Jednoduchý komunikační řetězec.

Komplikovanější případ nastává, pokud je komunikace mezi už. agentem a původním serverem vedena přes zprostředkovatelské uzly. Základními typy prostředníků jsou proxy servery, aplikační brány a tunely. Na obrázku 2.2 je uveden možný komunikační řetězec obsahující tři zprostředkovatelské uzly A, B a C, které jsou vloženy mezi uživatelského agenta a původní server. Dotaz nebo odpověď, která prochází celým řetězcem projde přes tři samostatné spojení. Tato skutečnost je důležitá z pohledu zpracování parametrů HTTP komunikace, kdy některé parametry jsou aplikovatelné pouze na spojení s nejbližším sousedem, některé pouze na koncové uzly řetězce nebo jsou aplikovatelné na všechny spojení v řetězci.



Obrázek 2.2: Komunikační řetězec s více uzly.

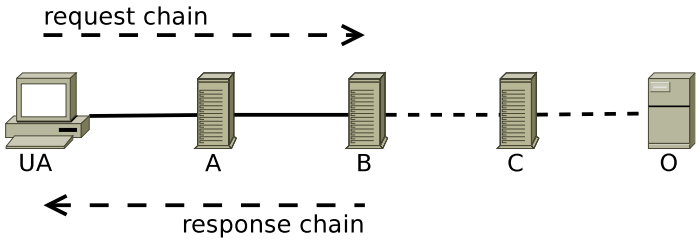
Uzly komunikačního řetězce, které nejsou tunely, mohou implementovat vlastní vyrovnávací paměti cache pro vyhodnocení příchozích dotazů. Výsledným efektem je zkrácený komunikační řetězec od už. agenta pouze po uzel s jednotkou cache, která obsahuje odpověď na zaslaný dotaz (na obrázku 2.3 se jedná o uzel B). Tímto je pro klienta UA zkrácena doba čekání na odpověď a také snížena zátěž uzlů C a O a síťových spojení mezi nimi.

2.1.3 Zprávy HTTP protokolu

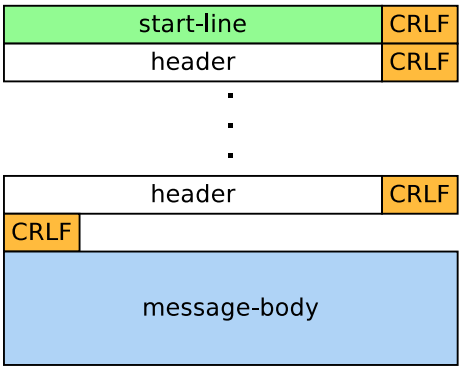
Zpráva je základní komunikační jednotkou protokolu HTTP, sestávající z úvodního řádku, *hlaviček*, prázdného řádku (sekvence `CRLF`) a případného těla zprávy (viz obrázek 2.4).

Hlavička sestává z názvu parametru a dvojtečkou oddělené hodnoty. Každá hlavička je na samostatném řádku ukončeném sekvencí `CRLF`.

Jak již bylo zmíněno u komunikačního modelu, protokol HTTP definuje dva typy zpráv a to *dotaz* (žádost) a *odpověď*.



Obrázek 2.3: Zkrácení komunikačního řetězce při použití HTTP cache.



Obrázek 2.4: Syntaxe HTTP zprávy.

Dotaz

Klient zasílá serveru dotaz, který obsahuje *dotazovací řádek* sestávající z identifikátoru metody, adresy požadovaného zdroje *URI* [7] a verze HTTP protokolu. Za tímto řádkem mohou následovat hlavičky zprávy. V případě protokolu HTTP 1.1 je povinná hlavička `Host` obsahující jmennou identifikaci internetového uzlu, který vlastní požadovaný zdroj. Tímto parametrem je jednoznačně určen zdroj v případě, kdy na jednom uzlu běží více HTTP serverů.

Adresa URI se může vyskytovat v jednom z následujících formátů:

absolutní URI – sestává z adresy serveru, absolutní cesty zdroje, dotazu a fragmentů (`http://www.domain.co/directory/file.php?q=1`).

absolutní cesta – definuje pouze absolutní cestu zdroje, není specifikována adresa serveru (`/directory/file.php?q=1`).

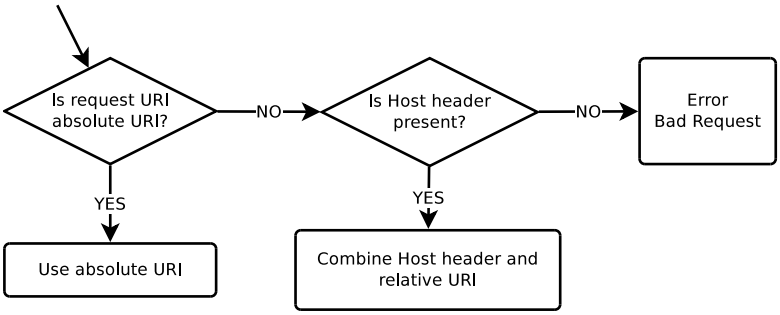
autorita – tento typ je použit v dotazech `CONNECT` a udává adresu serveru a číslo portu (`wwww.fit.vutbr.cz:80`).

***** – lze použít pouze pro dotazy typu `OPTIONS`, hvězda nespecifikuje konkrétní zdroj na serveru, ale celý server.

Určení absolutní URI je vyjádřeno diagramem 2.5.

Zpráva typicky obsahuje ještě další hlavičky, které mohou například upřesnit dotaz s ohledem na vlastnosti implementace klientského agenta.

Ukázka HTTP dotazu:



Obrázek 2.5: Určení adresy zdroje.

```
GET /pub/WWW/ HTTP/1.1\r\n
Host: www.w3.org\r\n
\r\n
```

Metody definované pro protokol HTTP 1.1 jsou *OPTIONS*, *GET*, *HEAD*, *POST*, *PUT*, *DELETE*, *TRACE* a *CONNECT*.

Metody GET, HEAD mohou být považovány za *bezpečné metody*, tedy jejich vykonání na straně serveru by nemělo mít žádný jiný význam než akce nalezení adresovaného zdroje. Vždy ale není možné zaručit, že server při provádění metody GET nevygeneruje vedlejší efekty (některé dynamické zdroje mohou přímo vyvolat akce porušující toto pravidlo).

Jinou vlastností metod GET, HEAD, PUT a DELETE je *idempotence*. Tato vlastnost zaručuje, že vícenásobné vyhodnocení identického dotazu obsahujícího některou z těchto metod má stejný výsledek jako vyhodnocení jediného dotazu – opakováním identických dotazů se nezmění stav serveru. Vyhodnocení metod OPTIONS a TRACE by nemělo způsobit žádné vedlejší efekty a tudíž lze tyto metody také označit za idempotentní.

Následuje stručný popis metod protokolu HTTP 1.1.

OPTIONS je metoda použitá pro zjištění dostupných možností komunikačního řetězce identifikovaného pomocí URI. Metoda umožňuje klientovi zjistit možnosti a případné požadavky asociované se zdrojem nebo dostupné funkce serveru.

GET je metoda určená pro získání jakékoliv informace (ve formě entity) identifikované pomocí URI. Pokud URI odkazuje proces produkující data, pak je tento proces spuštěn a v entitě odpovědi je vrácen výstup procesu a ne zdrojový text procesu.

HEAD je identická s metodou GET až na výjimku, která zakazuje serveru zaslat tělo zprávy v odpovědi. Server zašle odpověď, která má identické hlavičky jako v případě dotazu GET, ale neobsahuje žádné tělo. Použití metody je nejčastěji v získávání meta-informací o entitě, která by byla odeslána při použití GET.

POST je žádostí k serveru o akceptování entity obsažené v těle zprávy jako novou podřízenou instanci zdroje identifikovaného pomocí URI. Metoda je používána pro zaslání dat od klienta na server. Adresovaný zdroj je v tomto případě proces, který zpracuje entitu obsaženou v těle zprávy. Tímto zdrojem může být například data-zpracující proces nebo brána do jiného protokolu. Metoda *není* idempotentní.

PUT je žádost o uložení zasílané entity na adresu specifikovanou pomocí URI. Pokud URI odkazuje na již existující zdroj, pak má metoda aktualizační sémantiku. URI zároveň

identifikuje v dotazu obsaženou entitu a server nesmí aplikovat žádost na jiný než adresovaný zdroj. Metoda *je* idempotentní.

DELETE je metoda požadující po původním serveru odstranění zdroje identifikovaného pomocí URI.

TRACE je metoda dovolávající se vytvoření vzdálené zpětné smyčky na aplikační vrstvě pro zasílaný dotaz. Poslední příjemce tohoto dotazu by na něj měl odpovědět zprávou se stavovým kódem 200 (OK) a zároveň do těla entity odpovědi vloží celou zprávu s přijatým dotazem. Tato metoda umožňuje klientovi zjistit v jaké podobě obdrží poslední příjemce jeho dotaz procházející přes různé uzly v komunikačním řetězci.

CONNECT je metoda rezervovaná pro použití ve spojení s proxy, která se může dynamicky přepnout do tunelového módu. Metoda je používána pro ustavení tunelu při zabezpečení komunikace pomocí TLS [14].

Odpověď

Server zasílá klientovi odpověď na jeho dotaz, která obsahuje stavový řádek sestávající z identifikátoru verze protokolu, *stavového kódu* a doplněného o textový řetězec popisující daný stav. Odpověď může dále obsahovat hlavičky s metadaty a tělo nesoucí případná data.

Stavový kód je trojmístné číslo informující o výsledku zpracování přijatého dotazu. Kódy lze rozdělit do pěti tříd podle hodnoty nejvýznamnější číslice:

1xx: informační – informuje o stavu provádění dotazu, dotaz byl přijat a následuje jeho zpracování;

2xx: úspěch – dotaz byl přijat, pochopen a úspěšně zpracován;

3xx: přesměrování – je potřeba provést další dotaz(y) pro dokončení žádosti (např. přesměrování na jinou adresu);

4xx: chyba klienta – chybná syntaxe dotazu, dotaz nemůže být vyhodnocen (např. nedostatečná oprávnění už.);

5xx: chyba serveru – správný dotaz nemohl být vyhodnocen kvůli vnitřní chybě serveru

Podrobný popis jednotlivých kódů není předmětem této práce a lze jej nalézt v [12]. Stavové kódy jsou rozšiřitelné a HTTP aplikace nemusejí chápat význam všech kódů. Po aplikacích je požadována správná interpretace jednotlivých tříd (rozlišení podle první číslice kódu). Příklad HTTP odpovědi je znázorněn obrázkem 2.6.

2.1.4 Hlavičky zpráv

Následující podkapitoly popisují některé vybrané hlavičky HTTP zpráv, jejichž význam je důležtiý z hlediska rekonstrukce webového provozu a které se mohou vyskytnout dále v textu práce. Přesnou specifikace všech hlaviček protokolu HTTP 1.1 lze nalézt v [12, kap. 14].

```
HTTP/1.1 301 Moved Permanently\r\n
Date: Mon, 02 Jan 2012 23:53:00 GMT\r\n
Server: Apache/2\r\n
Location: http://www.w3.org/\r\n
Cache-Control: max-age=21600\r\n
Expires: Tue, 03 Jan 2012 05:53:00 GMT\r\n
Vary: Accept-Encoding\r\n
Content-Length: 226\r\n
Connection: close\r\n
Content-Type: text/html; charset=iso-8859-1\r\n
\r\n
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
```

Obrázek 2.6: Ukázka HTTP odpovědi (tělo odpovědi je zkrácenou pouze na první řádek)

Délka zprávy

Určení délky těla přenášené entity a tedy i konce zprávy je nezbytné pro korektní čtení zpráv, jež mohou být bezprostředně za sebou zřetězeny.

Hlavička `Content-Length` slouží k popisu délky těla entity. Její hodnota vyjadřuje počet oktetů. Ovšem při použití kanálového kódování se určí délka těla zprávy z použitého kódování. HTTP odpověď navíc může být ukončena uzavřením spojení, diagram 2.7 shrnuje postup určení délky těla odpovědi.

V případě určení délky HTTP dotazu je situace jednodušší, neboť přítomnost entity musí být signalizována přítomností hlavičky `Content-Length` nebo `Transfer-Encoding`.

Kódování obsahu

Protokol HTTP umožňuje aplikování různých kódování na těla přenášených zpáv, aby se efektivněji využívalo přenosové pásmo. Kódování lze použít v rámci jednoho přenosového kanálu (hlavička `Transfer-Encoding`, platnost „hop-by-hop“) nebo jej aplikovat přímo na entitu (hlavička `Content-Encoding`, kódování je vlastností enitity a jeho platnost je mezi koncovými uzly).

Ve specifikaci HTTP/1.1 jsou definovány následující typy kódování [12, kap. 3.5, 3.6]:

chunked – speciální formát rozdělení proudu bajtů do skupin *chunks*, kde *chunk* sestává z identifikátoru délky v počtu oktetů a ze samotných dat. Ukončení proudu bajtů je realizováno prázdným *chunkem* s délkou 0.

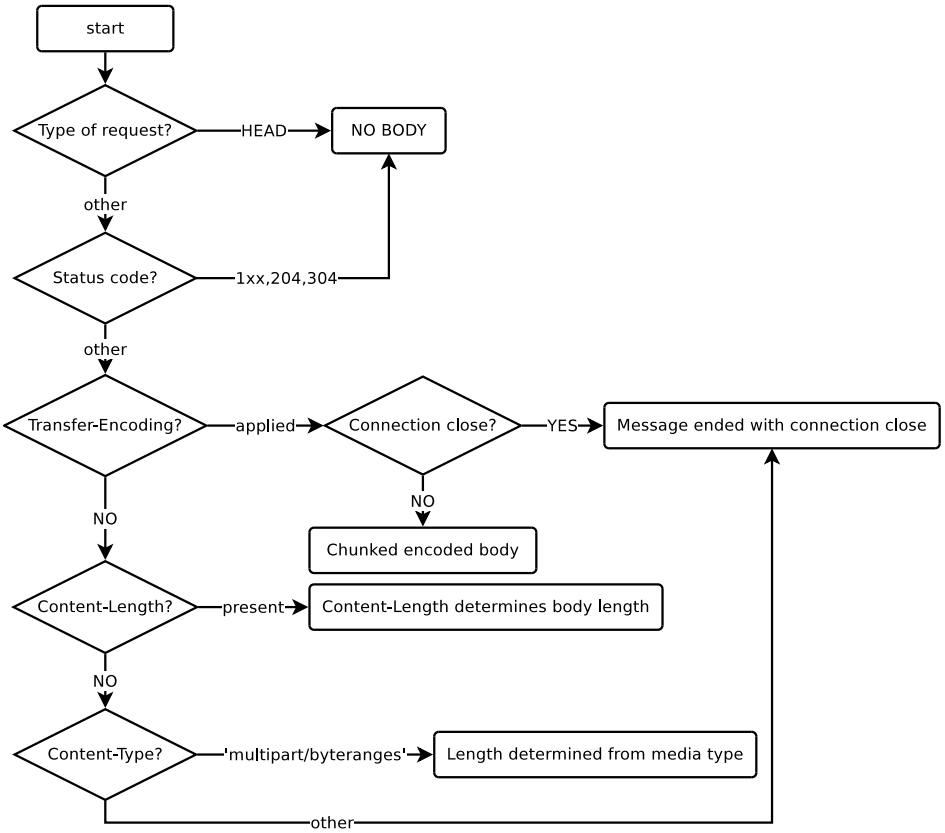
Toto kódování umožňuje efektivní přenos dynamicky generovaného obsahu spolu s informacemi nezbytnými pro příjemce, aby mohl detekovat přenos celé zprávy.

Typ kódování *chunked* lze aplikovat pouze jako kanálové kódování. Podle RFC 2616 musí všechny HTTP 1.1 kompatibilní aplikace podporovat toto kódování.

gzip – formát kódování kompresního programu `gzip`. Jedná se o Lempel-Ziv kódování (LZ77) s 32 bitovým kontrolním součtem popsané v [10].

compress – kódovnání odpovídá unixovému programu `compress`, Lempel-Ziv-Welch (LZW).

deflate – je kombinací formátů `zlib` a `deflate` popsaného v [9].



Obrázek 2.7: Určení délky těla odpovědi.

identity – výchozí, prázdné kódování, data zůstavají identická.

V rámci jedné zprávy lze aplikovat vícenásobné kódování, pak hlavička `Transfer-Encoding`, případně `Content-Encoding`, vystupuje jako zásobník, na jehož vrchol se vkládají postupně aplikovaná kódování. Vrchol zásobníku reprezentuje první (nejlevější) položka hlavičky.

Pokud bylo pro přenos zprávy aplikováno liboolné kanálové kódování odlišné od *identity* (hlavička `Transfer-Encoding`), pak poslední aplikované kódování musí být *chunked*. Tímto je zajištěna možnost přesného určení délky těla zprávy.

RFC 2616 dále uvádí, že každá HTTP/1.1 kompatibilní aplikace musí být schopna zpracovat *chunked* kódování [12, kap. 3.6.1].

2.2 Zabezpečení HTTP pomocí TLS

V počátcích zabezpečení HTTP komunikace pomocí SSL3 byla tato problematika řešena vytvořením nového URI schématu ('https') a vyhrazením portu 443. Paralelně tak mohly vedle sebe existovat nezabezpečená verze HTTP (URI schéma 'http' na portu 80) a zabezpečená verze pomocí SSL (URI schéma 'https' na portu 443). Podobně tomuto modelu se i pro další služby (např. SMTP, FTP) začaly přidělovat samostatná čísla *well-known* portů pro odlišení jejich zabezpečených verzí. Tento trend ovšem vedl k výraznému omezení dostupných well-known portů.

Uvedený problém je řešen v protokolu HTTP 1.1 pomocí přepnutí otevřeného HTTP spojení na použití TLS (Transport Layer Security). Tento postup je popsán v [14].

TLS vytváří zabezpečené *end-to-end* spojení, volitelně obsahující vzájemnou autentizaci obou koncových uzlů. Úvodní fáze využívá tři subprotokoly pro nastavení záznamové vrstvy, autentizaci uzlů, nastavení parametrů a hlášení chyb. Vytvořená vrstva je potom zodpovědná za šifrování, kompresi, a sestavení dat spojení. Tento proces je pro vyšší protokoly zcela transparentní.

Obrázek 2.8 shrnuje možnosti vytvoření tunelovaného spojení přepnutí spojení na zabezpečené. Zeleně podbarvené zprávy představují HTTP odpovědi, šedé pak HTTP dotazy.

2.2.1 Zabezpečené spojení vyžádané klientem

Klient může zažádat o použití zabezpečené komunikace se serverem prostřednictvím zaslání dotazu s hlavičkou `Upgrade: TLS/1.0`.

Požadavek na zabezpečenou komunikaci může být *volitelný*. V tomto případě klient o přepnutí může požádat během kteréhokoliv dotazu, jehož odpověď může být přenesena nezabezpečeně.

```
GET http://example.bank.com/acct_stat.html?749394889300 HTTP/1.1
Host: example.bank.com
Upgrade: TLS/1.0
Connection: Upgrade
```

Server na tento dotaz může odpovědět obvyklým způsobem (ignoruje požadavek na zabezpečení) nebo přepnutím do zabezpečeného módu (bude vysvětleno dále v textu).

Pokud klient vyžaduje použití TLS, zaslání odpovědi nezabezpečeným kanálem by bylo nepřípustné, pak musí iniciovat spojení zvláštním dotazem s metodou OPTIONS. Jedná se o *povinné* přepnutí spojení.

```
OPTIONS * HTTP/1.1
Host: example.bank.com
Upgrade: TLS/1.0
Connection: Upgrade
```

Pokud je server připraven na sestavení TLS spojení, pak je povinen odpovědět informační zprávou `101 Switching Protocol`, která musí obsahovat hlavičku `Upgrade` specifikující protokol, který bude použit.

```
HTTP/1.1 101 Switching Protocols
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
```

Server přepne na protokoly specifikované v hlavičce `Upgrade` bezprostředně po odeslání prázdného oddělovacího řádku odpovědi 101. Po úspěšné přepnutí spojení server musí zaslat odpověď na původní dotaz. Jakákoliv chyba při sestavování zabezpečeného spojení vede k ukončení stávajícího spojení.

2.2.2 Zabezpečení spojení vyžádané serverem

Podobně jako klient může server zažádat o přepnutí na zabezpečení spojení pomocí TLS. Na rozdíl od klienta ovšem server žádá prostřednictvím hlaviček HTTP odpovědi.

Varianta *volitelné* žádosti umožňuje serveru použít hlavičku `Upgrade` v jiné než 101 nebo 426 („Update Required“) odpovědi. Tímto dává najevo svůj požadavek na přepnutí na protokoly uvedené v hlavičce `Upgrade`.

V případě, že server bezpodmínečně vyžaduje použití zabezpečeného spojení pro dokončení klientova dotazu, pak server zašle chybovou zprávu `426 Upgrade Required`, která ve své hlavičce obsahuje požadovanou verzi TLS.

```
HTTP/1.1 426 Upgrade Required
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
```

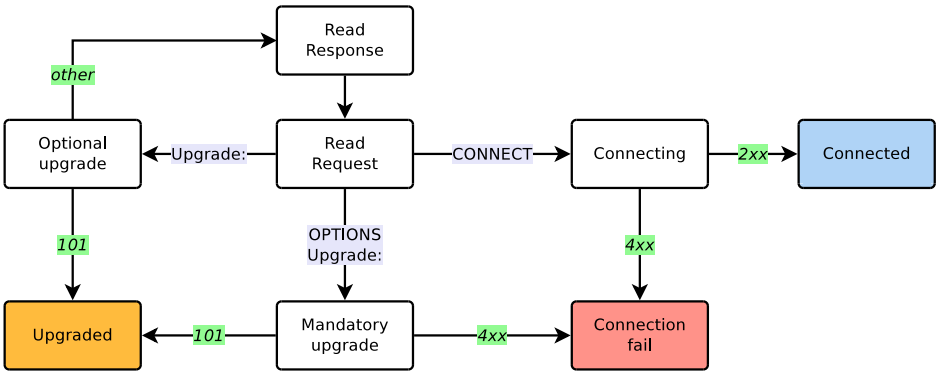
Ačkoliv může být klient ochoten přistoupit na sestavení zabezpečeného spojení, musí použít mechanismy uvedené v 2.2.1. Sestavení TLS spojení nemůže být zahájeno bezprostředně po přijetí odpovědi 426.

2.2.3 Přepnutí spojení při použití proxy

Hlavička `Upgrade` je aplikována na jednotlivá spojení mezi HTTP uzly. Jedná se o tzv. *hop-by-hop* hlavičku, jež je aplikovatelná pouze na nejbližší sousední HTTP uzel. Uživatelský agent, který zašle proxy dotaz s hlavičkou `Upgrade`, požaduje přepnutí protokolů pouze mezi jím samotným a proxy (nejedná se o end-to-end nastavení). TLS ovšem vyžaduje end-to-end konektivitu pro poskytnutí autentizace a pro zabránění útokům typu *man-in-the-middle*. V [14] je popsán koncept využívající zabezpečeného tunelového spojení, které lze sestavit pomocí žádosti typu `CONNECT`.

Prostřednictvím metody `CONNECT` žádá klient sestavení tunelového spojení jménem proxy.

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com:80
```



Obrázek 2.8: Vytvoření tunelu, změna spojení na šifrované.

Úspěšná odpověď (třída 2xx) na žádost typu `CONNECT` znamená, že proxy úspěšně sestavila spojení s požadovaným uzlem a přepnula klientovo spojení do tunelového módu se sestaveným spojením. Pokud je proxy připojena k požadovanému uzlu nepřímo přes další proxy, pak musí mezi těmito uzly rovněž proběhnout navázání tunelového spojení. Tento postup se opakuje, dokud není tunel navázán přímo se serverem.

Po ustavení tunelového spojení mezi už. agentem a původním serverem je typicky zahájeno přepnutí na zabezpečené spojení.

2.3 Rekonstrukce webového provozu

Na rekonstrukci webového provozu je v této práci nahlíženo jako na rekonstrukci HTML dokumentů. Jazyk HTML je hlavním značkovacím jazykem pro reprezentaci webových stránek.

Cílem rekonstrukce je získání a správná interpretace dat ze zachyceného síťového provozu, aby takto zpracovaná data mohla být snadno zpětně vizualizována. Součástí výstupu jsou metadata, které mohou být použity k analýze webového provozu. Ideálním výsledkem je zobrazení rekonstruovaných dokumentů (stránek) tak, jak je viděli původní uživatelé ve svých webových prohlížečích a maximální množství zjistitelných metadat, která lze podrobit statistickým analýzám.

Uvedenému ideálnímu cíli se lze přiblížit, ale ne ve všech případech ho lze dosáhnout. Hlavním problémem, se kterým se musí rekonstrukce síťového provozu vyrovnat, jsou chybějící nebo poškozená data.

V následující odstavcích je popsána idea rekonstrukce webového provozu z pohledu rekonstrukce HTML dokumentů. Princip je podobný činnosti webového prohlížeče, který na základě URL požadovaného zdroje žádá webový server o zaslání dat. Pokud jsou přijatá data HTML dokument, pak jej analyzuje, aby získal URL lokátory obsažených objektů. Zasláním dodatečných dotazů získá odkazované objekty a ty, společně se zdrojovým kódem HTML, použije pro sestavení finální podoby dokumentu.

Rekonstrukce je pasivní z hlediska síťové komunikace, rekonstrukční nástroj negeneruje žádný síťový provoz. Všechna potřebná data jsou získávána odposlechem komunikace původních zúčastněných stran. Rekonstrukční nástroj se snaží na základě analýzy provozu vytvářet kontext klientské aplikace, aby mohl správně interpretovat odposlechnutá data.

2.3.1 Předzpracování síťového provozu

Před samotným zpracováním HTTP protokolu je nezbytné provedení paketové analýzy (zpracování položek hlaviček protokolů pro dané pakety) a rekonstrukce toků transportního protokolu (v případě protokolu TCP lze použít například nástroj `tcpflow`) a každý takovýto tok je nutné jednoznačně identifikovat.

Dalším krokem je rozdělení takto získaných toků na jednotlivé zprávy aplikačního protokolu HTTP. Během analýzy hlaviček zpráv je zároveň možné provést export užitečných dat obsažených v tělech dotazů a odpovědí. Metadata popisují datový formát, použité kódování a velikost dat jsou přenášena v hlavičkách HTTP zpráv.

2.3.2 Párování HTTP dotazů a odpovědí

Základním úkolem při rekonstrukci webového provozu je párování HTTP dotazů a jim příslušejících odpovědí. Zpráva s dotazem nese ve svých hlavičkách informace o použité metodě (GET, HEAD, POST, ...), identifikaci požadovaného zdroje pomocí *URI* (Uniform Resource Identifier [7]) a případné další meta-informace. Odpověď pak obsahuje stavový kód, informující o stavu zpracování požadavku, a případné další hlavičky. Z dotazu je získána použitá metoda a identifikátor zdroje (URI), z odpovědi pak stavový kód. Takto získané dvojice zpráv jsou identifikovány pomocí URI z dotazu.

V protokolu HTTP verze 1.0 je pro každou dvojici dotaz – odpověď navázáno klientem samostatné TCP spojení [8, kap. 1.3]. Párování je v tomto případě triviální, v jednom spojení je vždy přenesen dotaz následovaný příslušející odpovědí. Protokol HTTP verze 1.1 implicitně navazuje *perzistentní spojení* [12, kap. 8.1], které umožňuje přenést více dotazů a jim příslušejících odpovědí v rámci jednoho spojení. Dále je umožněno zřetězení dotazů bezprostředně za sebou, aby se omezil nepříznivý vliv latence přenosu sítí. Odpovědi zasílané serverem pak musí být vždy odeslány v pořadí odpovídající přijatým dotazů. Párování je řešeno pomocí číslování pořadí dotazů a odpovědí.

2.3.3 Syntaktická analýza HTML kódu

První navrhovaná varianta rekonstrukce webového provozu vychází z myšlenky aktivního zpracování přenášených mediálních dat, které potom budou zobrazitelné pasivním vizualizačním nástrojem (pouze čtení lokálně uložených souborů). Rekonstrukce je v tomto případě prováděna jako syntaktická analýza kódu HTML dokumentů, které reprezentují stránky zobrazitelné ve webovém prohlížeči.

První fází je rozpoznání těchto dokumentů mezi entitami jednotlivých odpovědí. Následuje syntaktická analýza HTML kódu, jejímž výstupem je seznam referencí, které byly v kódu použity. Závěrečnou fází je přesměrování těchto referencí na odpovídající lokálně exportované soubory.

Výsledkem je upravený HTML dokument, jehož reference směřují pouze na lokálně exportované soubory s mediálními daty. Tento dokument lze následně zobrazit bez nutnosti dodatečného získávání obsažených mediálních dat prostřednictvím HTTP dotazů na původní zdrojové servery.

Návrhu a implementaci rekonstrukčního nástroje, který je založen na syntaktické analýze HTML kódu je věnována kapitola 3.

2.3.4 Využití proxy cache

Druhou uvažovanou variantou řešení rekonstrukce je použití HTTP proxy brány s vyrovnávací pamětí cache. Rekonstrukce v tomto případě spočívá v analýze HTTP zpráv a jejich uložení do vyrovnávací paměti cache. Proxy brána zpřístupňuje takto zrekonstruovanou HTTP komunikaci vizualizačnímu nástroji (klientský webový prohlížeč) prostřednictvím HTTP protokolu.

Toto řešení je složitější z pohledu nutnosti implementace proxy brány a jednotky cache. Výhodou je ovšem obecnější přístup k rekonstrukci webového provozu, kdy se zpracovávají čistě zprávy protokolu HTTP, a interpretace přenášených dat (např. HTML dokumentů) je ponechána na vizualizačním nástroji (běžný webový prohlížeč).

Kapitoly 4 a 5 jsou věnovány návrhu a implementaci rekonstrukčního nástroje využívajícího proxy bránu s jednotkou cache.

Kapitola 3

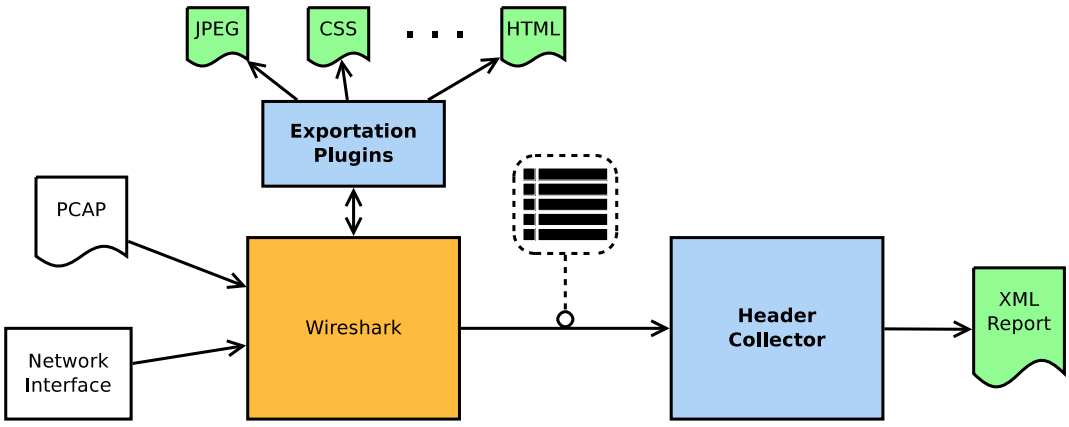
Rekonstrukce s vyžitím analyzátoru Wireshark

Kapitola je věnována návrhu a implementaci nástroje pro rekonstrukci webového provozu založeného na syntaktické analýze HTML dokumentů (popsána v kapitole 2.3.3). Tento nástroj byl implementován s využitím paketového analyzátoru *Wireshark*, zdrojové kódy řešení jsou dostupné na přiloženém CD nosiči.

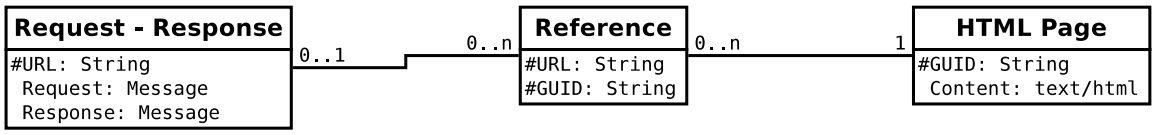
První podkapitola popisuje návrh nástroje, následující pokapitoly se již věnují implementaci jednotlivých částí a závěr kapitoly diskutuje dosažené výsledky.

3.1 Návrh rekonstrukčního nástroje

Architektura je postavená nad paketovým analyzátorem Wireshark, který předzpracovává vstupní síťový provoz. Rekonstrukce HTTP toků (párování dotazů a odpovědí) a syntaktická analýza HTML kódu je řešena *kolektorem hlaviček*. Blokové schéma architektury je znázorněno obrázkem 3.1.



Obrázek 3.1: Architektura využívající analyzátor Wireshark.



Obrázek 3.2: ER digram rekonstrukční databáze.

3.1.1 Paketový analyzátor Wireshark

Paketový analyzátor Wireshark je použit pro extrakci položek hlaviček síťových protokolů. Jeho vstupem je soubor se zachyceným síťovým provozem (PCAP soubory) nebo přímý vstup ze síťového rozhraní. Výstupem analýzy jsou metadata v podobě textové reprezentace hlaviček síťových protokolů a také soubory obsahující užitečná data přenášená protokolem HTTP (těla HTTP zpráv).

Analyzátor Wireshark byl zvolen s ohledem na jeho multiplatformnost, otevřené zdrojové kódy pod licencí GPL verze 2 a analýzu širokého spektra protokolů včetně HTTP.

Analýza je prováděna vzestupně ode dna protokolového (TCP/IP) zásobníku linkovou vrstvou počínaje a aplikační vrstvou konče. Wireshark je schopen rekonstruovat TCP toky a tyto pak může rozdělit na jednotlivé zprávy protokolu aplikační vrstvy (např. HTTP). Díky této vlastnosti analyzátoru je ulehčena implementace rekonstrukčního nástroje, protože extrakce položek hlaviček protokolů aplikační vrstvy je řešena při analýze. Kolektor pouze zpracovává takto získaná data bez nutnosti samostatně řešené rekonstrukce TCP toků a parsování HTTP zpráv.

Pro Wireshark byl navržen modul, který umožňuje export z interní datové reprezentace paketů ve Wiresharku do souborů dostupných přes souborový systém. Tento exportní modul je pak využíván zásuvnými moduly pro různé datové formáty (JPEG, PNG, zdrojové kódy CSS, HTML, JavaScript aj.) tvořícími nadstavbu nad modulem HTTP. Zapojení těchto modulů je patrné z obrázku 3.1.

3.1.2 Kolektor hlaviček síťových protokolů

Rekonstrukce HTTP provozu na úrovni toků HTTP zpráv je prováděna kolektorem hlaviček. Vstupem je textová reprezentace položek hlaviček protokolů exportovaných analyzátorem. Výstupem je zpráva ve formátu XML obsahující meta-informace o rekonstruovaných datech. Pro usnadnění zpětné vizualizace rekonstruovaných HTML dokumentů jsou vytvořeny kopie dokumentů s přepsanými referencemi směřujícími na lokálně uložené soubory s exportovanými daty.

Činnost kolektoru odpovídá popisu uvedenému v kapitole 2.3. Pro ukládání dvojic, referencí a HTML dokumentů byla navržena jednoduchá databáze, jejíž ER diagram je znázorněn obrázkem 3.2.

Tabulka párů obsahuje dvojice zpráv dotaz – odpověď. Řádky jsou indexované podle URI dotazů.

Tabulka referencí propojuje tabulku stránek a tabulku párů na základě referencí URI. Tabulka obsahuje reference z HTML dokumentů získané pomocí syntaktické analýzy.

Tabulka stránek ukládá datové objekty reprezentující HTML dokumenty, které jsou jednoznačně identifikované pomocí GUID.

Tabulky jsou postupně plněny v průběhu zpracovávání HTTP zpráv. Databáze kontroluje naplnění závislostí mezi tabulkou stránek a tabulkou párů prostřednictvím tabulky referencí. Jakmile mají všechny reference z dané stránky přiřazeny odpovídající páry podle URI, pak je rekonstrukce stránky dokončena. Následuje vložení záznamu o stránce do výstupní zprávy a posléze odstranění stránky a jejích referencí z tabulek databáze.

Záznamy z tabulky párů se neodstraňují, protože mohou být použity opakovaně, tabulka tak napodobuje funkci vyrovnávací paměť cache. Tato vlastnost umožňuje částečné vyrovnání se problémem chybějících dat, kdy příčinou může být právě použití interní cache webového prohlížeče nebo přístup už. agenta do Internetu přes proxy cache. Pokud má uživatelský agent požadované objekty dostupné ve své lokální cache, pak o ně nemusí žádat původní server a data nejsou přenášena sítí a tak nemohou být ani odposlechnuta.

3.2 Implementace

Následující podkapitoly popisují implementaci rekonstrukčního nástroje s využitím paketového analyzátoru Wireshark. První část je věnována implementaci rozšiřujících modulů Wiresharku, druhá část pak kolektoru hlaviček.

Výsledný nástroj sestává ze dvou aplikací, paketového analyzátoru *Tshark* (konzolová verze Wireshark) a kolektoru hlaviček. Spuštění analyzátoru se správnými parametry je provedeno automaticky z kolektoru. Předpokládá se hlavně automatizované použití nástroje bez uživatelské interakce, a proto není implementováno grafické uživatelské rozhraní. Možnost změny výchozích nastavení běhu nástroje je řešena předáváním parametrů při spuštění kolektoru.

3.3 Rozšíření analyzátoru Wireshark

Paketový analyzátor Wireshark, je implementován v jazyce C s využitím knihovny GLib, verze s grafickým UI využívá navíc knihovny GTK+. V současné době jsou implementována rozšíření pro analyzátor verze 1.6.0, zdrojové kódy analyzátoru jsou k dispozici na stránkách projektu [5] v sekci *Download*, odkud byly také pro implementaci převzaty.

Ve vývojářské dokumentaci projektu Wireshark [16] lze nalézt alespoň základní informace o architektuře analyzátoru a popis jeho vnitřní činnosti. Na obrázku 3.3 je znázorněno schéma funkčních bloků analyzátoru včetně vyznačených rozšíření, které byly implementovány.

Dumpcap – provádí zachytávání síťového provozu ze síťového rozhraní. Může využít knihoven *WinPcap* nebo *libpcap*.

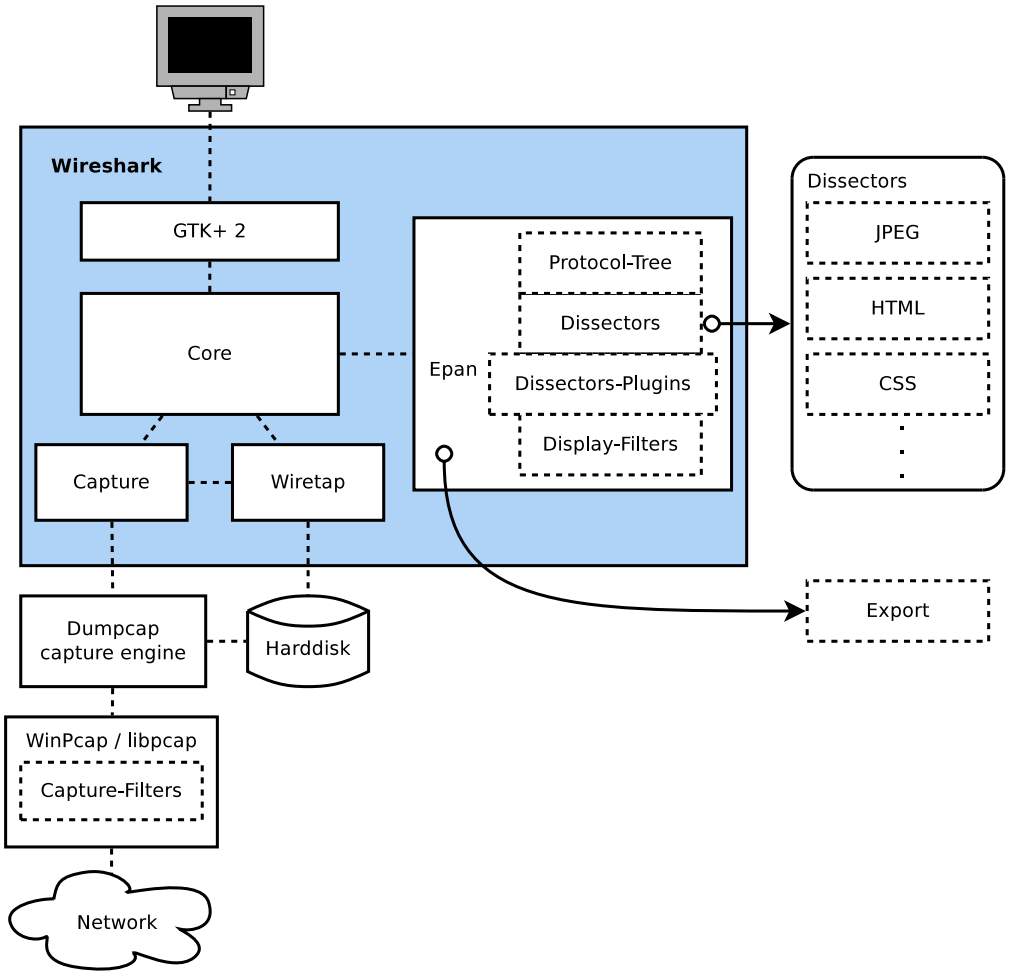
Capture – tvoří abstraktní rozhraní nad jednotkou Dumpcap.

Wiretap – je knihovna umožňující čtení a zápis souborů zachyceného síťového provozu ve formátu `libpcap` a dalších.

Epan – *Ethereal Packet Analyzer*, provádí analýzu paketů. V rámci práce byl rozšiřován pouze tento modul.

Core – propojuje ostatní funkční bloky.

GTK+ 2 – spravuje grafické už. rozhraní.



Obrázek 3.3: Blokové schéma analyzátoru Wireshark (převzato z [16]).

Detailní popis funkčních bloků není předmětem této práce a lze jej najít v [16] nebo ve zdrojových kódech analyzátoru.

Jak již bylo zmíněno v návrhu architektury, tak analyzátor Wireshark provádí rekonstrukci toků protokolů transportní vrstvy a také analýzu hlaviček zpráv aplikačních protokolů. Každý paket je analyzován samostatně a tvoří jeden záznam na výstupu. Navíc jsou postupně sestavovány zprávy aplikačních protokolů, které pak mohou tvořit vlastní záznamy na výstupu. Například jedna HTTP zpráva rozdělená na 3 IP pakety vytváří na výstupu 3 záznamy. První dva záznamy odpovídají prvním dvěma přijatým IP paketům, které obsahují pouze část HTTP zprávy. Poslední záznam pak obsahuje informace o třetím přijatém paketu a zároveň celou sestavenou HTTP zprávu.

3.3.1 Modul `export.h`

Je hlavním modulem pro export dat ze síťového provozu. Jeho rozhraní umožňuje analyzátorům jednotlivých datových formátů exportovat data z vnitřní reprezentace Wiresharku do souborů v souborovém systému. Tento modul je zodpovědný za unikátní pojmenovávání všech exportovaných souborů. Unikátní názvy souborů jsou řešeny pomocí počítadla souborů.

Vstupem exportovací funkce je ukazatel typu `void *`, který ukazuje na paměťový blok obsahující data k exportu, velikost datového bloku a ukazatel na znakový řetězec s požadovanou příponou exportovaného souboru. Funkce provede zápis bloku dat do souboru, jehož jméno je jedinečné a je odvozeno z počítadla souborů.

Pro každý exportovaný soubor je rovněž k interní datové struktuře zastupující HTTP zprávu přidána informace o názvu a umístění souboru. Tyto informace jsou pak součástí textového výstupu, který tvoří vstup kolektoru hlaviček.

Pomocí parametrů lze nastavit cílový adresář, kam budou exportované soubory zapisovány. Lze také zakázat export do souborů, výstup rekonstrukce je pak tvořen pouze XML zprávou.

3.3.2 Moduly datových formátů

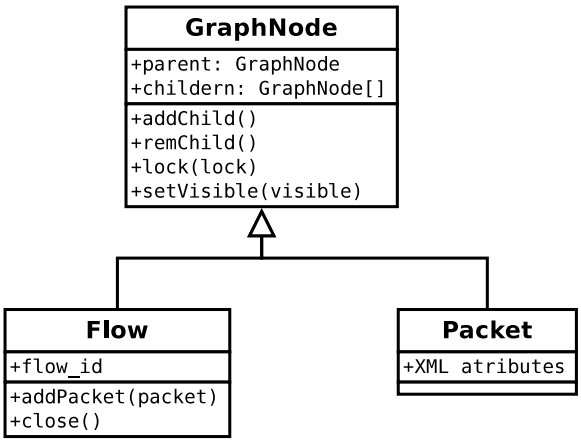
Moduly analyzující protokoly nižších vrstev automaticky předávají data zapouzdřených protokolů analyzátorům vyšších vrstev. Například pokud analyzátor protokolu IP zjistí, že hlavička `Protocol` obsahuje hodnotu 6, pak předá obsah těla paketu ke zpracování analyzátoru protokolu TCP. Analyzátory datových formátů představují virtuální vrstvu nad aplikační vrstvou HTTP.

Rozpoznání typu přenášených dat v těle HTTP zprávy je možné na základě hodnoty hlavičky `Content-Type`. Analyzátor HTTP vyhledá v tabulce registrovaných protokolů ten analyzátor, který odpovídá MIME typu v hlavičce `Content-Type` a předá mu data ke zpracování.

Rozhraní pro analyzátory je definováno ve zdrojových kódech Wiresharku. Vnitřní implementace je triviální, je zde pouze volána exportní funkce z modulu `export.h`.

V současné době je podporován export datových formátů JPEG, PNG, BMP, GIF, ICO, CUR, HTML, CSS, JavaScript, XML, SWF (Adobe Shockwave Flash). Rozšíření na další formáty je jednoduše realizovatelné přidáním dalších modulů.

Analyzátor Tshark je spuštěn s parametry, které nastavují jeho textový výstup jako jednotlivé hlavičky oddělené znakem `\t`, kde každý paket je reprezentován jedním řádkem. Všechny požadované hlavičky jsou v parametrech vyjmenovány a pořadí hlaviček na



Obrázek 3.4: Digram tříd

výstupu odpovídá jejich pořadí v parametrech.

3.4 Kolektor hlaviček HTTP

Implementace kolektoru vychází z návrhu uvedeného v kapitole 3.1.2 a také z modelu zpracování paketů ve Wiresharku. První část této kapitoly je věnována zpracování protokolových hlaviček, ve druhé části je popsána implementace rekonstrukce HTML dokumentů.

Pro implementaci kolektoru byl zvolen programovací jazyk Python a to kvůli bezproblémové přenositelnosti, velké škále dostupných knihoven a úspornému kódu. Kolektor je napsán pro verzi Python 2.7. Jako syntaktický analyzátor HTML zdrojových kódů je použita knihovna `libxml2dom` [1].

3.4.1 Zpracování hlaviček protokolů

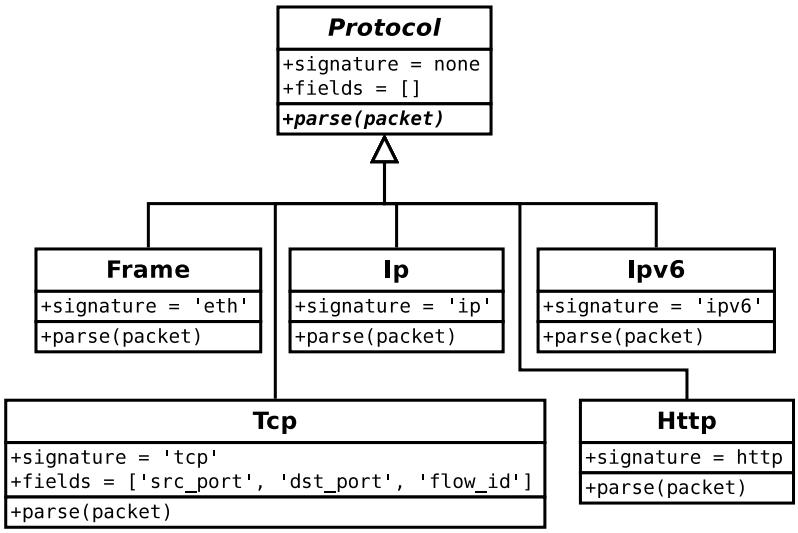
Vstupem kolektoru jsou textově reprezentované hlavičky protokolů, které extrahoval analyzátor Wireshark, kde každý paket je zastoupen jedním řádkem. Každý takový řádek obsahuje položku, která specifikuje všechny zastoupené protokoly v tomto paketu (např. `eth:ip:tcp`). Třída parser na základě těchto protokolových *signatur* postupně předává paket konkrétním analyzátorům. Všechny tyto analyzátory jsou registrované v asociativní tabulce protokolů `ProtoTable` a indexované pomocí signatur (viz atribut `signature` na obrázku 3.5).

Třída `Packet`

Každý vstupní řádek je interně reprezentován třídou `Packet` (viz obrázek 3.4), která jednak ve svých atributech obsahuje všechny hlavičky paketu a zároveň tvoří listový uzel ve výstupním XML stromu.

Třída `Protocol`

Každý protokol, jehož hlavičky mají být zpracovávány je reprezentován samostatnou třídou odvozenou od bázové třídy `Protocol` (viz obrázek 3.5). Odvozené třídy musí implementovat



Obrázek 3.5: Digram tříd protokolů.

metodu `parse`, která dostane na vstupu objekt třídy `Packet` reprezentující jeden vstupní řádek. Tato metoda pak provádí analýzu v rámci daného protokolu na základě hlaviček paketu.

Třída `Protocol` zajišťuje automatickou registraci v tabulce protokolů a také registraci požadovaných hlaviček (atribut `fields`, pro přehlednost uveden jen u třídy `Tcp`).

3.4.2 XML strom

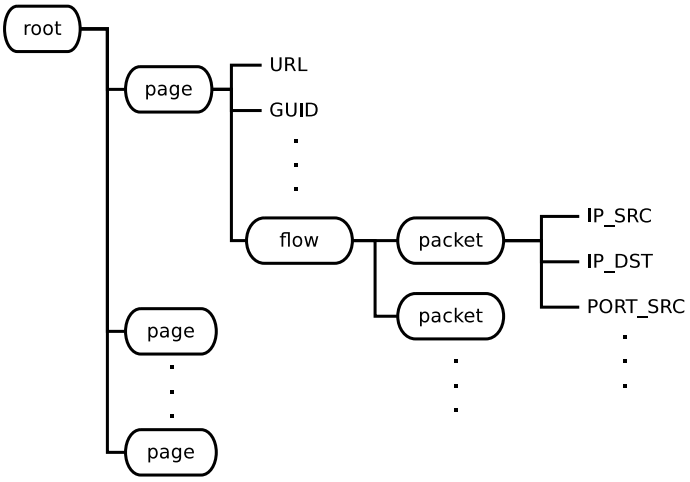
Výstupní zpráva je formátu XML. Vnitřní reprezentace XML zprávy je realizována pomocí stromu. Třída `Packet` představuje listové uzly stromu. Analyzátory jednotlivých protokolů modifikují tento koncový uzel přidáváním XML atributů.

Uzel paketu může být v rámci stromu přesouván. Jednotlivé třídy protokolů mohou do stromu přidávat další uzly a tímto vytvářet hierarchickou strukturu. Příklad takového stromu je uveden na obrázku 3.6, non-terminální uzly jsou označeny oválem, terminální uzly (textové elementy XML) jsou pojmenovány velkými písmeny.

Uzly stromu jsou reprezentovány třídou `GraphNode`, která poskytuje rozhraní pro základní grafové operace jako přidání a odebrání uzlu, vyhledání uzlu podle kritérií. Další metody umožňují skrývání uzlů nebo celých podstromů ve výstupním XML (podle požadované úrovně detailů), případné zamykaní uzlů proti přesunu.

Dále byla implementována třída `Flow` (viz digram tříd na obrázku 3.4), odvozená od `GraphNode`, která poskytuje rozhraní pro jednoduchou správu paketových toků. Tento uzel lze po ukončení toku uzamknout pro přidávání dalších paketů a zabránit tak změnám v uzlech potomků. Díky této optimalizaci je tok prohlášen za uzavřený a může být zapsán do výstupní XML zprávy a uvolněn z paměti.

Podle vytvořeného stromu je pak sestavena výstupní XML zpráva, kde jednotlivé XML elementy odpovídají uzlům stromu.



Obrázek 3.6: Příklad stromové reprezentace XML elementů.

3.4.3 Rekonstrukce HTML dokumentů

Rekonstrukce HTML dokumentů vychází z databáze popsané v návrhu architektury uvedeného v kapitole 3.1.2.

Byly implementovány třídy tabulek `PairTable`, `UrlTable` a `PageTable`. Bázovou třídou pro tabulky je třída `Table` (viz diagram tříd na obrázku 3.7).

Třída `Table`

Tato třída poskytuje základní rozhraní pro správu tabulky jako je přidávání, odebírání záznamů, získávání záznamů podle jejich klíče. Odvozené třídy musí pouze implementovat privátní metodu `get_id(data)`, která vrací klíč pro vkládaná data pomocí metody `insert(data)`.

Rozhraní třídy je rozšířeno o možnost propojení tabulek pomocí signálů a slotů (metody `connect()` a `emit()`). Odvozená třída tabulky může pomocí privátní metody `emit()` signalizovat svoji změnu (např. přidání nebo smazání řádku) a tato změna je signalizována všem slotům připojeným pomocí metody `connect()`. Použití signálů bylo motivováno možností použití více nezávislých tabulek, které mohou být pomocí signálů násobně propojeny.

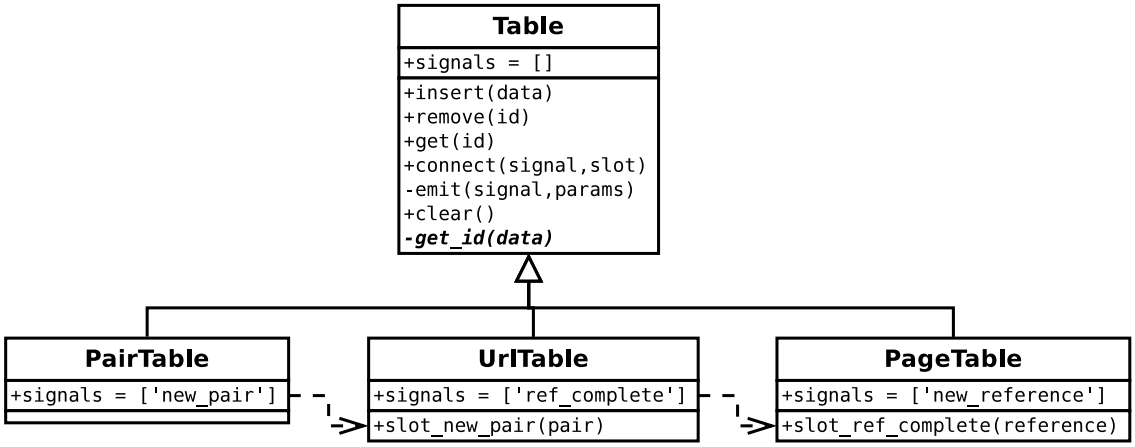
Činnost databáze

Z níže uvedeného popisu funkce jednotlivých tabulek je patrná činnost databáze. Implementace tabulek vychází z návrhu architektury uvedeného v kapitole 3.1.2.

PairTable – ukládá dvojice zpráv dotaz – odpověď (třídy `Packet`). Řádky tabulky jsou indexované podle URL lokátoru dotazu. Všechny příchozí pakety jsou uloženy do této tabulky. Vnitřní implementace tabulky zaručuje správné přiřazené odpovědí k dotazům tak, jak bylo uvedeno v návrhu v kapitole 3.1.2.

V případě, že je pár kompletní, je emitován signál `new_pair` informující o této události.

V případě, že vkládaná zpráva obsahuje HTML dokument, je tato zpráva také vložena do tabulky `PageTable`.



Obrázek 3.7: Digram tříd tabulek

UrlTable – uchovává všechny reference z HTML dokumentů. Ukládané záznamy (třídy `HttpUrl`) jsou dvojice sestávající z URL reference, která je indexem tabulky, a identifikátoru dokumentu, ze kterého byla reference získána.

Slot `slot_new_pair` zachycuje signál `new_pair` z tabulky `PairTable`. Tímto je signalizováno naplnění reference – k požadovanému URL existuje HTTP odpověď s požadovanými daty. Takto vyřešená referenční závislost je signalizována signálem `ref_complete`.

PageTable – ukládá záznamy o HTML dokumentech. Před vložení zprávy obsahující HTML dokument je provedena syntaktická analýza. Získané reference jsou vloženy do tabulky `UrlTable`. Výstupem analýzy je DOM objekt reprezentující HTML dokument.

Přijetí signálu `ref_complete` indikuje naplnění reference daného dokumentu. Taková reference může být v DOM objektu přepsána na lokální umístění exportovaného datového souboru. Při vyřešení všech referencí daného dokumentu je zapsán podstrom odpovídající dokumentu do výstupní XML zprávy. Dále je zapsána na disk nová verze HTML dokumentu z DOM objektu s přesměrovanými referencemi a následně je záznam o dokumentu včetně jeho referencí odebrán z tabulek databáze.

Diagram tříd databázových tabulek na obrázku 3.7 znázorňuje pomocí přerušovaných čar propojení signálů a slotů v rekonstrukční databázi.

3.5 Vlastnosti implementace

Činnost navržené architektury je závislá na analyzátoru Wireshark. Nevýhodou tohoto modelu je značná složitost analyzátoru, který je primárně určen pro paketovou analýzu co nejširšího spektra síťových protokolů. Rekonstrukce HTTP provozu naproti tomu vyžaduje stavové zpracování toků – a to pouze pro protokol TCP na transportní vrstvě a pro HTTP protokol aplikační vrstvy. Komplexnost Wiresharku rozšiřuje chybovou doménu rekonstrukčního nástroje. Analyzátor Wireshark je neustále vyvíjen, což přináší nutnost aktualizace zdrojových kódů tvořících rozhraní pro exportní moduly.

Implementovaný rekonstrukční nástroj byl testován na základní množině dat, která obsahovala webovou komunikaci s webmailovými servery (Seznam, Yahoo, Gmail, Hotmail

aj.). Dalším vzorkem byla množina statických webových prezentací (bez asynchronní komunikace). Testovací data jsou dostupná na přiloženém CD. Zpětná vizualizace byla provedena webovým prohlížečem Mozilla Firefox verze 9.0.1.

Výsledky rekonstrukce byly velmi závislé na obsažených elementech v HTML dokumentech. Použití dynamických prvků jako jsou skriptovací jazyky (např. JavaScript) a kaskádové styly není schopna pouhá analýza HTML kódu zpracovat. Dalším problémem nástroje je neschopnost se vypořádat s asynchronně přenášenými daty.

Uvedená omezení pramenící z nutnosti interpretovat už. data přenášená v HTTP zprávách vedla k přehodnocení přístupu k rekonstrukci webového provozu, což vyústilo v navržení a implementování nového nástroje založeného na použití proxy brány s vyrovnávací pamětí cache.

Kapitola 4

Návrh rekonstrukčního nástroje

Tato kapitola je věnována návrhu rekonstrukčního nástroje využívajícího HTTP proxy cache, jenž má být náhradou za řešení založené na analyzátoru Wireshark popsané v kapitole 3. V úvodní části je představena platforma rekonstrukční skupiny v rámci projektu Sec6net, jejíž součástí má být vyvíjený nástroj. Následuje popis samotného návrhu architektury rekonstrukčního nástroje. V závěru jsou shrnuty vlastnosti navrhované architektury.

Výstupem této práce má být nástroj, který umožní rekonstrukci webového provozu. Formou rekonstrukce v tomto případě rozumějme zpracování HTTP zpráv a analýzou jejich prezentaci. Cílem je, aby výsledný pohled na zrekonstruovaná data co nejpřesněji odpovídal pohledu uživatele, který byl původním iniciátorem rekonstruované komunikace.

Snahou je poskytnout řešení, které bude co nejvíce obecné a umožní tak rekonstrukci webového provozu různých aplikací vystavených nad protokolem HTTP. Těmto požadavkům odpovídá architektura využívající HTTP proxy cache. Nástroj je zaměřen pouze na zpracování zpráv protokolu HTTP a interpretace obsahu zpráv, který je aplikačně závislý, je ponechána na klientském webovém prohlížeči.

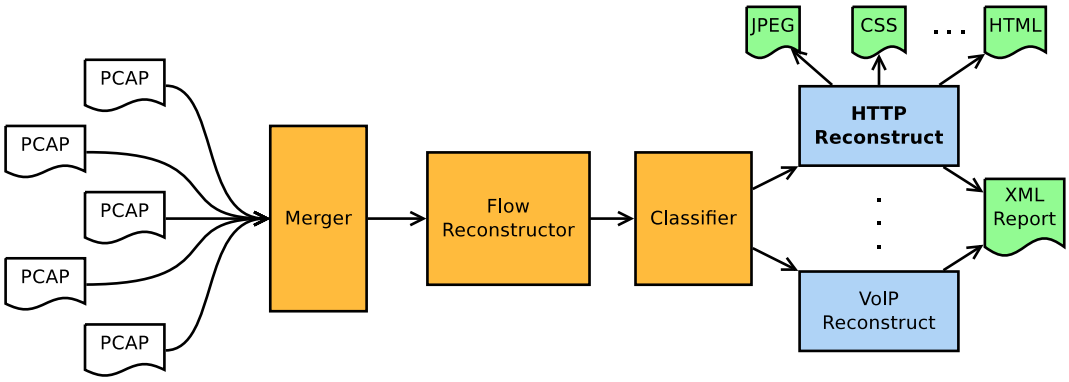
4.1 Platforma rekonstrukční skupiny

Návrh nové architektury rekonstrukčního nástroje vychází z použití jednotné platformy pro rekonstrukci aplikačních protokolů rekonstrukční skupiny projektu Sec6net. Tato platforma si klade za cíl poskytnout abstraktní vrstvu služeb nad vstupními síťovými daty. Nástroje této platformy budou provádět předzpracování síťového provozu podobně, jako analyzátor Wireshark u předchozí architektury, ale bez zpracování aplikační vrstvy. Předzpracování proběhne pouze na úrovni rekonstrukce toků protokolů transportní vrstvy.

Blokové schéma nového návrhu je znázorněno obrázkem 4.1.

Merger je jednotka spojující více PCAP souborů do jednoho souvislého datového proudu. Jedním z požadavků na novou architekturu je zpracování zachyceného provozu, který je rozdělen do více menších souborů. Rozdělení na menší soubory je nezbytné z důvodu rozložení zátěže mezi více fyzických disků při odposlechu síťové komunikace na vysokorychlostních linkách.

Flow Reconstructor pak provádí rekonstrukci toků protokolů transportní vrstvy (obdobně jako unixový nástroj `tcpflow`). Jeho výstupem jsou datové soubory obsahující rekonstruované datové proudy jednotlivých toků a také soubor obsahující meta-informace o těchto tocích (IP adresy, čísla portů, časové značky aj.).



Obrázek 4.1: Nová architektura

Classifier je klasifikační jednotkou, která se snaží rozpoznat jaký typ komunikace představuje daný tok a ty pak předává ke zpracování nástrojům pro konkrétní aplikační protokoly.

Výstupem předzpracování jsou soubory, které obsahují rekonstruované toky transportní vrstvy a XML zpráva obsahující meta-informace k těmto tokům. Formát XML je výhodný z hlediska snadné rozšiřitelnosti a přenositelnosti.

4.2 Rekonstrukce webového provozu

Řešení rekonstrukce vychází z použití HTTP proxy brány s lokální vyrovnávací pamětí cache, tak jak je popsáno v 2.3.4. Navrhovaný nástroj implementuje proxy bránu včetně vyrovnávací paměti, přičemž interpretace obsahu zpráv je ponechána na uživatelském agentovi – internetovém prohlížeči.

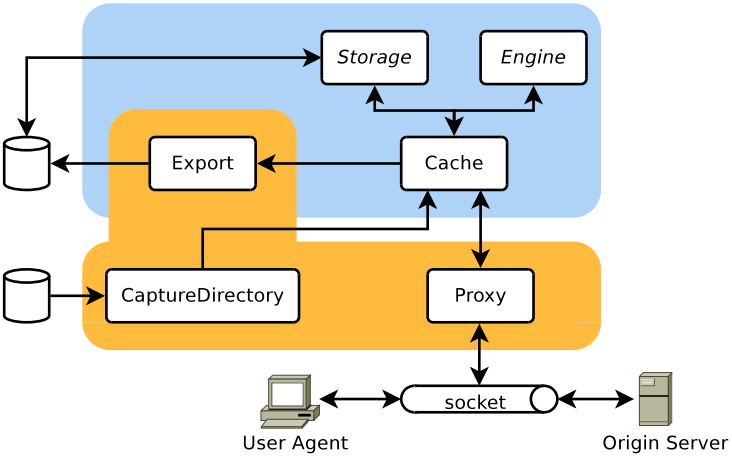
Nejdůležitější součástí nástroje je jednotka `Cache`. Jejím hlavním cílem je poskytovat odpovědi na dotazy prohlížeče provádějícího vizualizaci rekonstruovaného provozu tak, aby byl co nejpřesněji napodoben kontext původního klienta, který inicioval tuto rekonstruovanou komunikaci.

Obrázek 4.2 představuje blokové schéma nástroje a z něj patrné i jeho použití pro rekonstrukci. Barevné označení přirovnává funkčnost bloků k architektuře využívající analyzátor Wireshark znázorněné obrázkem 3.1.

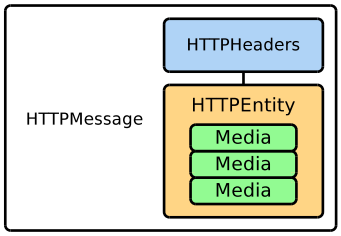
4.2.1 Zprávy komunikačního protokolu

Výměna informací mezi moduly navržené architektury je realizována zasíláním zpráv, kde komunikační protokoly jsou popsány přenášenými datovými objekty. Použití níže popsaných datových typů je patrné z obrázků 4.6, 4.5 a 4.7.

`ByteStream` zapouzdřuje datový proud, poskytuje rozhraní pro čtení a zápis dat, zjištění aktuální pozice čtecí a zápisové hlavy a její posuv, příznak konce proudu. Představuje jednotné rozhraní nad různými datovými objekty jazyka Python (sokety, soubory, paměťové bloky).



Obrázek 4.2: Blokové schéma rekonstrukčního nástroje.



Obrázek 4.3: Zanoření tříd reprezentujících atributy HTTP zpráv.

`HTTPMessage` je abstraktní třída zapouzdřující HTTP zprávu. Odvozené třídy `HTTPRequest` a `HTTPResponse` poskytují rozhraní pro čtení a editaci atributů zpráv (např. verze protokolu, návratový kód, dotazovací metoda).

`HTTPHeaders` je atributem třídy `HTTPMessage` a zapouzdřuje hlavičky HTTP zprávy. Její rozhraní umožňuje čtení a editaci hlaviček a jejich hodnot.

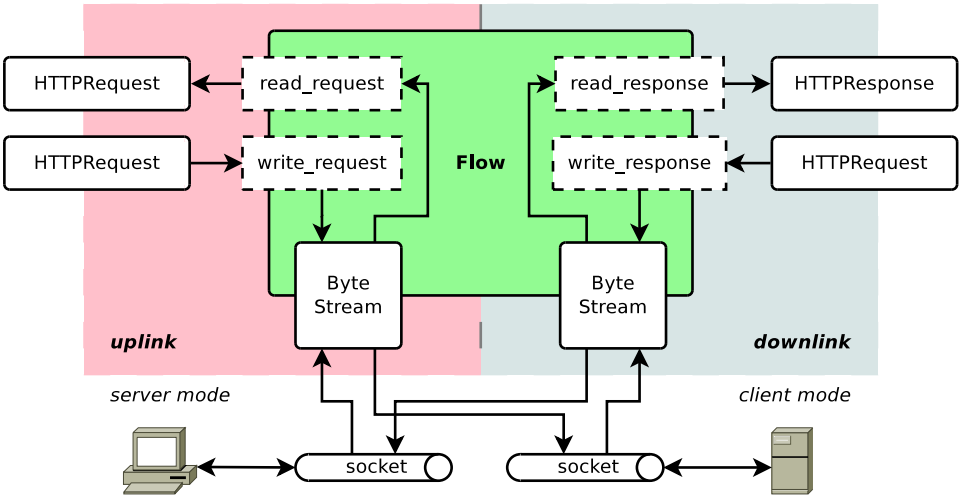
`HTTPEntity` zastupuje HTTP entitu, umožňuje její kódování, dekódování a zpřístupňuje data, která jsou přenášena v jejím těle. V rámci jedné entity může být přenášeno více médií. Obrázek 4.3 objasňuje zanoření výše uvedených tříd.

`Flow` reprezentuje tok protokolu HTTP mezi dvěma uzly (klient–server). Poskytuje rozhraní pro sekvenční čtení a zápis zpráv (znázorněno obrázkem 4.4). Tento datový kanál může být otevřen v jednom ze tří režimů:

pasivní režim umožňuje pouze čtení dotazů a odpovědí. Je vytvořen pro statická data získaná z PCAP souborů. Zastupuje zároveň klienta i server.

server je režim toků, které jsou vytvořeny na základě příchozího spojení od klienta (lokálně zastupuje server). Umožňuje číst dotazy a zapisovat odpovědi.

klient – slouží pro připojení k serveru (lokálně zastupuje klienta). Umožňuje zapisovat dotazy a číst odpovědi.



Obrázek 4.4: Rozhraní třídy `Flow`.

Instance třídy `Flow` jsou základními datovými jednotkami komunikace modulů `Proxy`, `CaptureDirectory` a `Cache`.

`Media` zastupuje media přenášená v těle entity (obrázek, HTML kód, aj.). Tato datová jednotka je exportovatelná. Pro každý datový typ může být vytvořena samostatná třída, která může implementovat pokročilé analýzy nad užitečnými daty (např. jazykovou analýzu HTML kódu).

`Record` je datovým typem záznamu v perzistentní databázi modulu `Cache`. Obsahuje metadata HTTP toků a zpráv nezbytná pro činnost cache.

FlowReader

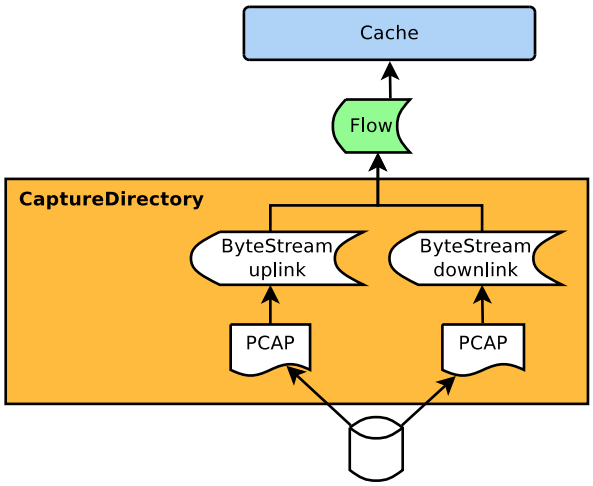
Výše popsaná třída `Flow`, která zastupuje komunikační kanál protokolu HTTP, poskytuje pouze základní rozhraní pro čtení a zápis zpráv, kdy jsou interpretovány pouze hlavičky nezbytné pro určení délky zprávy. Veškerá režie spojená s udržováním perzistentního spojení, vyjednávání typu přenášeného obsahu (např. použití kódování) a přepnutím spojení na šifrovaný kanál je ponechána na implementaci ve třídách pracujících s instancemi třídy `Flow`.

Pro usnadnení práce s HTTP komunikačními kanály byla navržena třída `FlowReader`, která bude abstrahovat spojení mezi klientem a serverem na úrověň výměny entit 2.1.1. Vnitřní implementace pak zajistí výběr použitého přenosového kódování, udržování perzistentní relace spojení, případně opětné navázání přerušeného spojení.

4.2.2 Modul `CaptureDirectory`

Tento modul tvoří rozhraní pro zpracování zachytaného síťového provozu. Zpracovaná statická data jsou použita pro naplnění HTTP cache a jsou z nich exportována obsažená multimedia. Blokové schéma na obrázku 4.5 znázorňuje činnost modulu z jeho zapojení s modulem `Cache`.

Soubory typu PCAP obsahující síťovou komunikaci jsou předzpracovány standardním unixovým nástrojem `tcpflow`, který provede rekonstrukci provozu na úrovni transportní



Obrázek 4.5: Zapojení modulu `CaptureDirectory`.

služby TCP. Pro každý TCP tok, identifikovaný zdrojovou a cílovou IP adresou a zdrojovým a cílovým číslem portu, jsou vytvořeny dva soubory. Jeden soubor obsahuje sekvenci oktetů tvořící jednosměrnou komunikaci mezi klientem a serverem; druhý soubor pak reprezentuje opačný směr. Soubory jsou pojmenovány podle schéma `srcIP:srcPort-dstIP:dstPort`.

Adresář obsahující výše uvedené dvojice souborů je vstupem modulu `CaptureDirectory`. Modul provede sestavení dvojic souborů na základě pojmenování souborů tak, aby každý výsledný pár představoval jeden TCP tok a tedy i jeden HTTP tok. V jednotlivých dvojicích se určí typ obou souborů jako *uplink*, komunikační kanál pro HTTP dotazy (klient–server), nebo *downlink*, přenos HTTP odpovědí (server–klient).

Každý rozpoznaný soubor je otevřen instancí `ByteStream` a z dvojic těchto datových kanálů jsou vytvářeny instance `Flow` (režim *pasivní*), které jsou výstupem modulu.

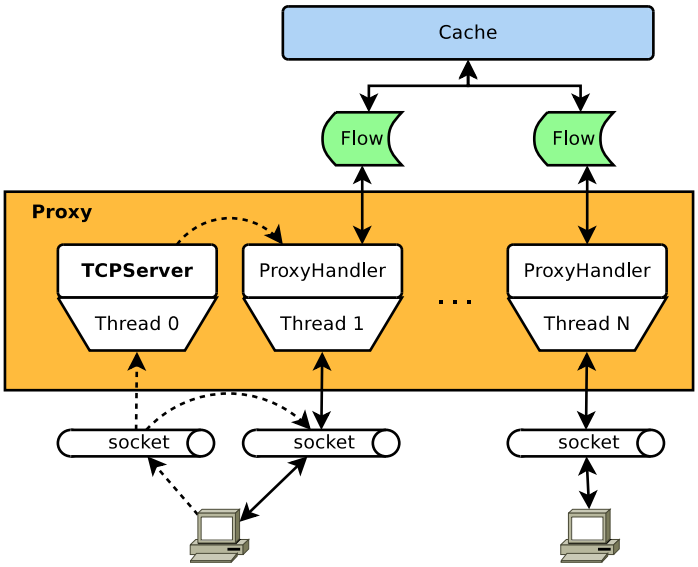
4.2.3 Modul `Proxy`

Brána vystupuje jako *non-transparent* proxy, ke které se připojuje už. agent (webový prohlížeč) provádějící vizualizaci rekonstruovaného provozu.

Poskytuje rozhraní nad síťovou vrstvou aplikačních soketů jazyka Python, které je využíváno modulem `Cache`. Tvoří druhý vstupní adaptér rekonstrukčního nástroje. Na rozdíl od `CaptureDirectory` nejsou vstupní data statická, ale jedná se o aktivní komunikační kanály propojující rekonstrukční nástroj s klientským webovým prohlížečem i se zdrojovými servery.

Vstup tvoří příchozí spojení od klienta, pro něž se vytvoří dva datové kanály typu `ByteStream`, kdy jeden zpřístupňuje čtení a druhý zápis z/do aplikačního soketu. Tyto kanály modul zapouzdří do instance třídy `Flow` (režim *server*). Druhým možným vstupem je požadavek ze strany `Cache` na spojení ke zdrojovému serveru. Pro takový požadavek je navázáno spojení na síťové vrstvě, následované vytvořením kanalů `ByteStream` následně opět zapouzdřených do instance třídy `Flow` (režim *klient*). Oba případy vysvětluje obrázek 4.4.

Aby byla zajištěna obsluha vícenásobných současných klientských požadavků je modul implementován jako konkurentní TCP server. Pro každé příchozí spojení je alokováno samostatné vlákno, které provede obsluhu požadavku. Způsob zapojení modulu včetně znázor-



Obrázek 4.6: Činnost modulu `Proxy`.

nění obsluhy příchozího spojení ilustruje schéma 4.6 (přerušovaná čára symbolyzuje postup navazování příchozího spojení od klienta).

4.2.4 Modul `Cache`

Hlavní modul nástroje, který zpracovává příchozí toky generované vstupními adatptéry. Řídicí logika je přesunuta do samostatného modulu `Engine`, který může existovat ve více implementacích podle požadované funkcionality nástroje ((ne)povolení proxy, (ne)povolení exportu).

Pro každý vstupní tok je vytvořena vlastní instance řadiče `Engine`, aby mohlo být obslouženo více požadavků současně.

`Engine`

Implementuje řídicí logiku zpracování HTTP toku.

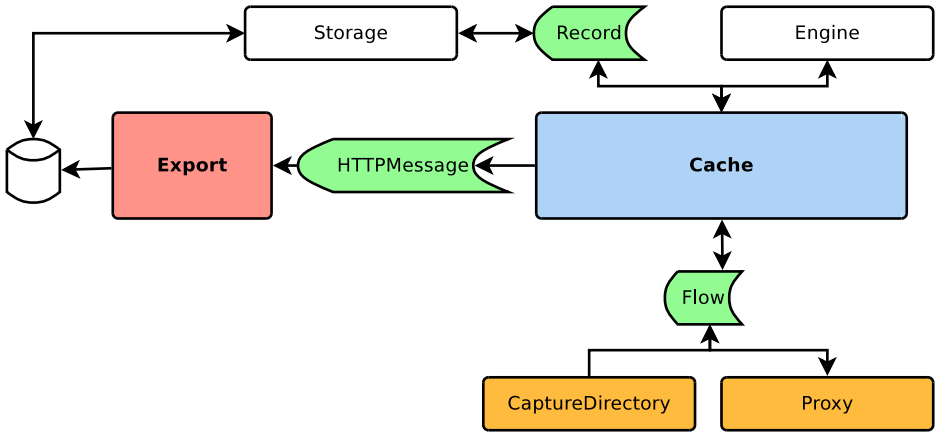
Základní úlohou řadiče je správa záznamů v úložišti, což představuje vyhledání záznamu pro příchozí dotaz od klienta a aktualizace záznamu při přijetí odpovědi od zdrojového serveru.

Diagramy znázorňující činnost řadiče při vyhledávání nebo vkládání záznamu do úložiště jsou na obrázcích 5.11 a 5.10, podrobnější popis k těmto alogritmům se věnuje kapitola 5.5.11.

`Storage`

Představuje perzistetní úložiště metadat nezbytných pro činnost řadiče `Engine`. Ukládané datové objekty jsou typu `Record` a mohou obsahovat libovolná data, která do nich řadič vloží. Úložiště vystupuje jako asociativní paměť adresovaná textovými řetězci.

Z důvodu možného vícenásobného přístupu a zápisu dat do sdíleného úložiště je pro každé vlákno zajištěn exkluzivní přístup pomocí zamykaní této datové struktury.



Obrázek 4.7: Zapojení modulu `Cache` a `Export`.

`PageHistory`

Tato třída popisuje datovou strukturu uchovávající historii rekonstruovaných webových stránek. Je vstupem prohlížeče, který provádí zobrazení rekonstruované webové komunikace. Rozhraní této třídy je dostupné z řídicího modulu `Enigne`, kde je řízeno vkládní záznamů.

Na obrázku 4.7 není tato třída zobrazena jako samostaný modul, ale je součástí třídy `Cache`. Perzistentní uložení historie je zajištěno pomocí modulu `Storage`.

4.2.5 Modul `Export`

Modul provádějící export medií do souborů v lokálním úložišti. Vstupem jsou objekty typu `HTTPMessage`.

Pro exportovaný záznam je určena cesta k výstupnímu adresáři (např. podle doménového jména zdrojového serveru) a data jsou poté zapsána do souboru. Po úspěšném zápisu je aktualizován záznam o cestě k exportovanému souboru v atributech média.

4.3 Vlastnosti architektury

Návrh nové architektury vychází ze zkušeností získaných během implementace a testování původního řešení využívajícího analyzátor Wireshark. Použití HTTP proxy cache představuje robustnější řešení, které se odprostilo od interpretace přenášených dat a maximálně se věnuje samotnému protokolu HTTP.

Při tvoření návrhu byl kladen důraz na škálovatelnost nástroje, aby mohly být postupně měněny a přidávány dílčí funkční bloky a výsledná aplikace se tak přizpůsobovala měnícím se požadavkům na rekonstrukci webového provozu. Například změna řízení jednotky cache nebo přidání nového vstupního adaptéru, který umožní zpracování vstupních dat v jiném formátu než PCAP soubory. Aby bylo dosaženo této modularity, tak bylo věnováno značné úsilí návrhu rozhraní funkčních bloků.

Výzvou pro novou architekturu je rekonstrukce provozu aplikací, které využívají asynchronní komunikace. Předcházející implementace nebyla schopna řešit tento typ provozu, což značně omezovalo možnosti jejího reálného využití.

Kapitola 5

Implementace

Následující kapitola je věnována popisu implementace rekonstrukčního nástroje – HTTP proxy brány s vyrovnávací pamětí cache podle požadavků představených v kapitole 4. V úvodní části je zdůvodněna volba implementačního prostředí, dále jsou popsány režimy činnosti nástroje, následované popisem implementovaných modulů a tříd, jejichž návrh je popsán v přechozí kapitole. V závěru jsou diskutovány vlastnosti implementace.

5.1 Volba prostředí

Pro implementaci rekonstrukčního nástroje byl zvolen jazyk Python verze 3.2, konkrétně jeho nejrozšířenější implementace *CPython*. Výhodou zvoleného jazyka je jeho vynikající přenositelnost, protože se jedná o jazyk interpretovaný virtuálním strojem, a také velmi obsáhlá standardní knihovna poskytující široké spektrum nástrojů, jež jsou v rámci projektu použity. Další předností je bezesporu rychlost vývoje aplikací psaných v jazyce Python (zejména díky vysoké míře abstrakce a úspornému kódu), který se takto dobře hodí i pro sktiptování rozsáhlých aplikací.

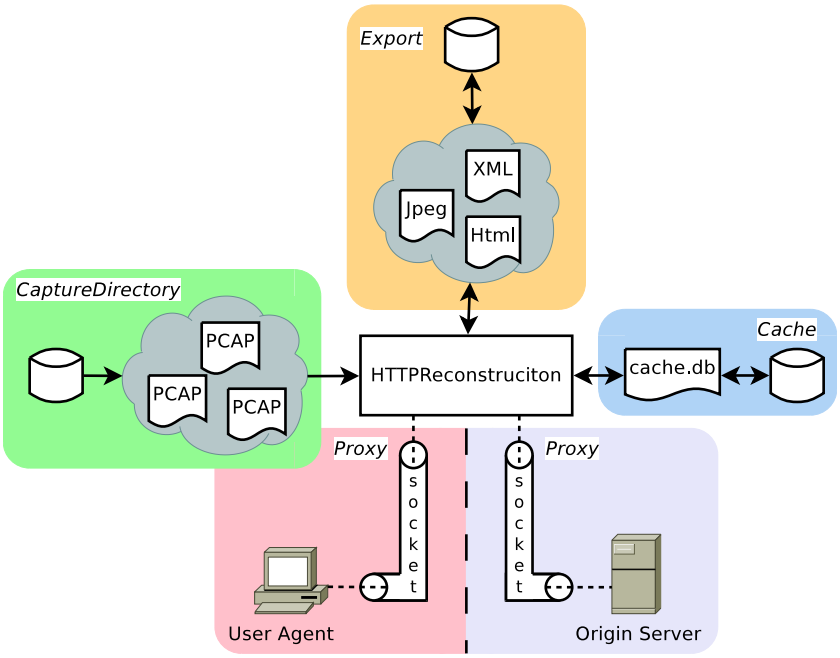
Interpretace virtuálním strojem přináší také své nevýhody co se týče časové a paměťové složitosti. Výkon nástroje je tedy velmi ovlivněn právě volbou jazyka, ale z pohledu této práce není výkon kritickým parametrem. Zpracování síťového provozu probíhá zpětně ze staticky uložených dat a běh v reálném čase je uvažován pouze u proxy brány. Předpokádá se, že počet uživatelských agentů, kteří budou připojeni přes bránu, je v řádu jednotek a časová odezva brány tak nebude výrazně narůstat.

5.2 Režimy činnosti nástroje

Implementovaná aplikace může být spuštěna v různých režimech činnosti podle dostupnosti vstupních dat a požadavků na výstup. Obrázek 5.1 znázorňuje celkový diagram užití, vysvětlení jednotlivých případů následuje dále v textu.

5.2.1 Čtení vstupních dat

Základním požadavkem na implementovaný nástroj je zpracování zachyceného síťového provozu, ze kterého je rekonstruována komunikace HTTP protokolu. Modul CaptureDirectory, na obrázku 5.1 znázorněn zcela vlevo, zajišťuje čtení vstupních dat a předává je k dalšímu zpracování.



Obrázek 5.1: Ilustrace režimů činnosti rekonstrukčního nástroje

V současné době podporovaný formát odpovídá výstupním souborům unixového nástroje `tcpflow`, který provádí rekonstrukci síťového provozu na úrovni transportní vrstvy, konkrétně pro spojovanou službu TCP.

Pokud je parametrem předána cesta k adresáři se vstupními data, jsou tato zpracována jako první a až poté je případně spuštěna proxy brána.

5.2.2 Proxy brána

Tvoří základní součást celého nástroje, neboť s ní komunikuje uživatelský agent provádějící vizualizaci rekonstruovaných dat. Brána může pracovat ve dvou režimech, přičemž první z nich umožňuje pouze komunikaci mezi lokálním klientem a bránou, brána v tomto případě vystupuje jako server. Tento případ je na obrázku znázorněn v levém dolním rohu.

V druhém režimu je navíc povolena komunikace brány s externími servery, brány tedy vystupuje jako klient. Toto je užitečné, pokud vstupní data rekonstrukce nejsou úplná nebo jsou zastaralá a je žádoucí jejich dodatečné získání nebo aktualizace. Více o možnosti přístupu k externím zdrojům dat pojednává kapitola věnovaná popisu implementace proxy brány (5.5.10) a jednotky cache (5.5.11). Tento případ užití je znázorněn v pravém dolním rohu.

Pokud není požadována vizualizace rekonstruovaných dat, nemusí být proxy brána vůbec spuštěna (např. při předzpracování dat a jejich vložení do cache pro pozdější vizualizaci).

5.2.3 Použití jednotky cache

Jednotka cache tvoří hlavní funkční blok implementované aplikace. Řídicí algoritmy pro její správu řídí činnost celého nástroje, tedy export dat a komunikaci s proxy bránou. Jejím

základním úkolem je správa databáze odpovědí, které jsou použity pro odbavení příchozích dotazů od klientů.

Tato jednotka je vždy přítomna, ale změnou řídicího modulu cache je možné dosáhnout požadovného chování nástroje, tedy i vypnutí správy cache (použije se prázdný řídicí modul). Více o řídicích modulech a činnosti jednotky cache je popsáno v kapitole 5.5.11. Na diagramu 5.1 je použití cache vyobrazeno zcela vpravo.

Parametrizovat lze umístění souboru s perzistentním úložištěm a také volbu řídicího modulu.

5.2.4 Export dat

Dalším z požadavků na výstup rekonstrukce webového provozu je možnost exportování dat, která jsou přenášena v tělech HTTP zpráv. Takto získaná uživatelská data jsou uložena do souborů odpovídajících formátů, díky čemuž jsou jednoduše dostupná v souborovém systému a mohou být později podrobena dalším analýzám.

Exportované soubory navíc představují zdroj dat pro pozdější vybavování požadavků klientů připojených k proxy bráně. Data již nemusí být uložena v rámci perzistentního úložiště jednotky cache, ale mohou být zpětně přečtena z exportovaných souborů, což značně snižuje velikost úložiště a zrychluje tak přístup k uloženým položkám. Na diagramu 5.1 je tento případ užití znázorněn zcela nahoře.

Výše popsané případy lze mezi sebou libovolně kombinovat použitím příslušných parametrů spuštění pro dosažení požadovaného chování nástroje. Popis parametrů spuštění nástroje není důležitý z hlediska diplomové práce a lze jej nalézt v dokumentaci zdrojových kódů projektu na přiloženém CD.

5.3 Členění projektu na balíčky a moduly

Zdrojové kódy jednotlivých modulů jsou pro lepší přehlednost a snazší orientaci rozděleny do několika balíčků, které tvoří logicky uzavřené celky.

`my_http` – sdružuje moduly implementující třídy HTTP zpráv, entit a hlaviček;

`flow` – obsahuje moduly implementující třídy `Flow` a `FlowReader`;

`cache` – balíček zapouzdřující moduly tříd `Cache`, řídicích jednotek `Enigne` a persistentní úložiště `Storage`;

`proxy` – zahrnuje implementaci třídy `Proxy` a obslužné rutiny tříd `Handler`;

`network` – balíček vytvořený pro implementaci tříd reprezentujících síťovou vrstvu a to třídu `Packet` a `IPAddress`;

`media_types` – v rámci tohoto balíčku jsou implementovány třídy reprezentující jednotlivé typy mediálních dat;

`tools` – soustřeďuje pomocné třídy a funkce využívané ostatnímy moduly

Pro každý balíček byla vytvořena základní sada automatizovaných testů, moduly pojmenované `test.py`, které mají ověřit správnost implementace daného balíčku. Testování je věnována samostatná kapitola 6.1.

5.4 Implementace modulů

Zde je popsána implementace významných modulů, které jsou součástí balíčku `main`. Ostatní moduly jsou popsány v rámci kapitoly 5.5, která je věnována detailnějšímu popisu implementace tříd.

Modul `main.py`

Tento modul tvoří vstupní bod rekonstrukční aplikace, je přímo spouštěn interpretem jazyka Python. Jeho činností je zpracování předávaných parametrů, sestavení konfigurace a podle požadavku spouští dílčí výpočetní moduly.

Modul `config.py`

Aby byla zajištěna maximální přizpůsobitelnost chování rekonstrukčního nástroje je snahou umožnit detailní konfiguraci jednotlivých modulů. Možným řešením je vytvoření globální instance třídy `Config`, která bude obsahovat všechny konfigurovatelné parametry. Tímto bude zajištěna dostupnost parametrů všem modulům.

Modul `config.py` definuje třídu `Config` a zároveň vytváří její jedinou, globálně dostupnou instanci `CONFIG`. Nastavení hodnot parametrů této instance je provedáděno pomocí funkcí implementujících zpracování parametrů příkazové řádky, které využívají modul standardní knihovny `argparse`. Výhodou tohoto centralizovaného přístupu je snadná rozšiřitelnot a přehlednost všech konfigurovatelných parametrů.

Popis významu některých parametrů bude vysvětlen dále v kapitole u popisu tříd a lze jej také nalézt v programové dokumentaci, která je součástí zdrojového kódu.

Modul `codings.py`

Prokotol HTTP umožňuje použití různých typů kódování aplikovatelných na těla zpráv tak, aby se efektivněji využilo přenosové pásmo (viz 2.1.4). Tento modul poskytuje funkce umožňující kódování a dekódování dat formátů *chunked*, *gzip* a *deflate*.

Funkce v tomto modulu implementují rozhraní pro standardní knihovny `gzip` – kódování *gzip* a `zlib` – kódování *deflate*. Zpracování *chunked* kódování je implementováno přímo modulem `codings.py`.

Moduly balíčku `tools`

V průběhu implementace jednotlivých modulů bylo nutné řešit některé základní algoritmické úlohy, které se vyskytovaly v menších obměnách na různých místech v kódu. Soustředěním těchto dílčích úloh vznikl balíček `tools`, který poskytuje pomocné funkce a třídy pro ostatní implementované moduly.

`uri.py` – protokol HTTP využívá pro adresaci zdrojů lokátoru URI, který se může v hlavičkách zprávy a v dotazovacím řádku vyskytovat v několika různých formách 2.1.3. Převody mezi těmito formáty provádí funkce implementované v modulu `uri.py`. Ke své činnosti využívají standardního modulu `urllib.parse`, z nějž je také převzata interní reprezentace lokátoru – třída `SplitResult` (popsána v dokumentaci jazyka Python [3]).

`message.py` implementuje třídy odvozené od `HTTPResponse`, které zastupují chybové, případně informativní, HTTP zprávy zasílané proxy bránou připojenému klientovi, například *404 Not Found* nebo *502 Bad Gateway*. Zprávy obsahují vysvětlující informaci v podobě HTML dokumentu.

`flow.py` poskytuje funkci provádějící obousměrné tunelování dat mezi dvěma aktivními HTTP toky. Jedná se o asynchronní příjem a zasílání dat mezi dvěma sokety pomocí systémového volání `select`. Tunelové spojení je vytvořeno na základě žádosti klienta zprávou `CONNECT`.

Modul `test.py`

Hlavní testovací modul, z něhož jsou spouštěny dílčí testovací moduly jednotlivých balíčků. Provádí testování implementace celého nástroje na úrovni spouštění modulu `main.py`, což odpovídá reálnému použití.

Cílem je otestovat funkčnost nástroje pro různé konfigurace režimů činnosti, které byly popsány v 5.2. Hodnocení výsledků testů je prováděno na základě analýzy výstupních dat rekonstrukce jako jsou počty zpracovaných zpráv, počty a typy exportovaných mediálních dat a počty záznamů v úložišti jednotky `cache`.

Podrobnějšímu popisu použitých testovacích dat a získávání referenčních výsledků je věnována samostatná kapitola 6.1.

5.5 Implementace tříd

Následující podkapitoly se věnují detailnějšímu popisu implementace tříd, které tvoří základní stavební bloky výsledné aplikace. Hlavní pozornost je věnována rozhraní tříd, důležitým vlastnostem vnitřní implementace a návaznosti na okolní objekty. Přesný popis atributů tříd a parametrů metod není přemětem této práce a lze jej nalézt v dokumentaci zdrojových kódů projektu.

Použité názvy vestavěných funkcí, tříd a metod standardní knihovny jazyka Python odpovídají pojmenováním uvedeným v dokumentaci [3], kde lze také nalézt přesný popis a význam těchto pojmů.

5.5.1 Identifiable

Jedná se o obstraktní třídu, která poskytuje jedinou metodu `uid()`, která vrací náhodný 128 bitový identifikátor UUID odpovídající RFC 4122 ([17]). Identifikátor je získan voláním funkce `uuid4()` z modulu `uuid` standardní knihovny.

Jedinečnost identifikátoru je použita například pro jednoznačné pojmenování exportovaných souborů (třída `Media` je odvozena od třídy `Identifiable`) a pro odlišení objektů v ladících výpisech.

5.5.2 ByteStream

Tato třída poskytuje univerzální rozhraní pro vstupně výstupní operace prováděné nad souborovými objekty nebo síťovými sokety. Přístup je stále na nízké úrovni abstrakce přes datový typ `bytes`. Chybové stavy jsou ošetřeny vyvoláním příslušných výjimek. Obrázek 5.2 znázorňuje UML diagram této třídy.

V rámci celé implementace je pro datovou reprezentaci vstupů a výstupů datových kanálů použit typ `bytes`, který reprezentuje pole bajtů. Díky tomuto řešení nedochází k dezinterpretaci řětězců obsahujících znaky v různém kódování a sekvencí `CRLF` označujících konec řádků podle [12]. Datový typ `str` totiž uvažuje textové řetězce sestávající ze znaků (nikoliv jednotlových oktetů) v určitém kódování a také adaptuje symboly ukončení řádků pro danou platformu (Unix `LF`, Windows `CRLF`, Mac OS `CR`).

Otevření a uzavření datového proudu

Otevření datového proudu se provádí metodu `opens(src, mode)`, jejíž první parametr udává cestu k souboru v lokálním souborovém systému (textový řetězec typu `str`) nebo je předána reference na objekt typu `socket`. Druhým povinným parametrem je určen režim otevření a to buď právě čtení nebe právě zápis (datový proud je vždy otevřen pouze v jednom směru). Všechny otvírané souborové objekty jsou otevřeny v binárním režimu a to z důvodu přesné interpretace sekvencí ukončujících konce řádků jak bylo uvedeno výše.

V případě požadavku na přístup k lokálnímu souboru se provede jeho otevření voláním vestavěné funkce jazyka Python `open()` se a vrácená reference na souborový objekt je uložena do atributu `_fp`. Tento režim je použit pro čtení statických dat, která jsou zpracovávána vstupním adaptérem `CaptureDirectory`.

Pokud je předávaným parametrem reference na objekt typu `socket`, pak se předpokládá, že tento soket zastupuje otevřený TCP kanál. Implementace třídy `socket` ze standardní knihovny poskytuje medotodu `makefile()`, která vytvoří virtuální souborový objekt nad síťovým soketem. Přes takto vytvořený objekt lze k soketu přistupovat stejně jako k souboru pomocí blokujících metod `read()`, `readline()` a `write()`. Reference na původní soket je uložena do atributu `_socket` a virtuální soubor je uložen do `_fp`.

Uzavření datového proudu je provedeno na základě volání metody `close()`. V případě, že byl otevřen soubor z lokálního souborového systému, je tento zavřen standardním voláním `close()`. V případě zapouzdření síťového soketu je nejprve zavřen virtuální soubor, a poté je uzavřeno spojení soketu metodou `shutdown()` (uzavírá se pouze určitý *směr* spojení podle režimu oteření datového proudu).

Operace čtení a zápisu

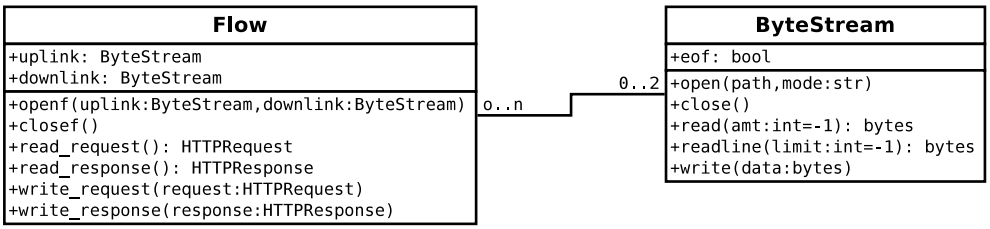
Pro čtení a zápis dat jsou implementovány metody `read(amt=-1)`, `readline(limit=-1)` a `write(data)`. Jejich funkčnost odpovídá stejnojmeným metodám třídy `BufferedIOBase` z knihovny `io`. Tyto metody jsou blokující. Použitelné jsou pouze operace podle režimu otevření datového proudu (buď pouze čtení, a nebo pouze zápis).

U metod pro čtení lze parametrizovat maximální počet oktetů, jež mají přečteny. V případě, že bylo dosaženo konce souboru (nebo bylo uzavřeno spojení soketu), je vrácen prázdný řetězec `b''`. Parametrem metody `write()` je reference na pole oktetů typu `bytes`, které mají být zapsány.

Volání těchto metod je vnitřně delegováno na objekt odkazovaný atributem `_fp`.

5.5.3 Flow

Tato třída je abstrakcí HTTP spojení mezi klientem a serverem. Přístup k tomuto toku je na vyšší úrovni abstrakce, kdy základní datovou jednotkou čtení a zápisu jsou HTTP zprávy. Třída je znázorněna UML diagramem 5.2.



Obrázek 5.2: Diagram třídy `Flow` a `ByteStream`.

Přístup k datovým zdrojům je realizován přes rozhraní dvou instancí `ByteStream`, kdy jedna představuje kanál *uplink* (spojení klient → server) a druhá *downlink* (server → klient). Různé režimy činnosti budou vysvětleny v následující části.

Otevření a uzavření toku

Otevření toku provádí metoda `openf(uplink, downlink, mode)`, jejíž první dva povinné parametry typu `ByteStream` určují použité kanály *uplink* a *downlink*. Třetím povinným parametrem je určení režimu otevření, jehož možné varianty shrnuje následující výčet.

pasivní – HTTP kanál vystupuje současně jako klient i server, umožňuje čtení dotazů i odpovědí, zápis zpráv není povolen. V tomto režimu jsou otevírány toky reprezentující statická vstupní data.

klient – instance vystupuje jako klient a vzdálená připojená strana je server, je umožněn zápis dotazů (*uplink*) a čtení odpovědí (*downlink*). Tento kanál je otevřen pro spojení s externími zdrojovými servery.

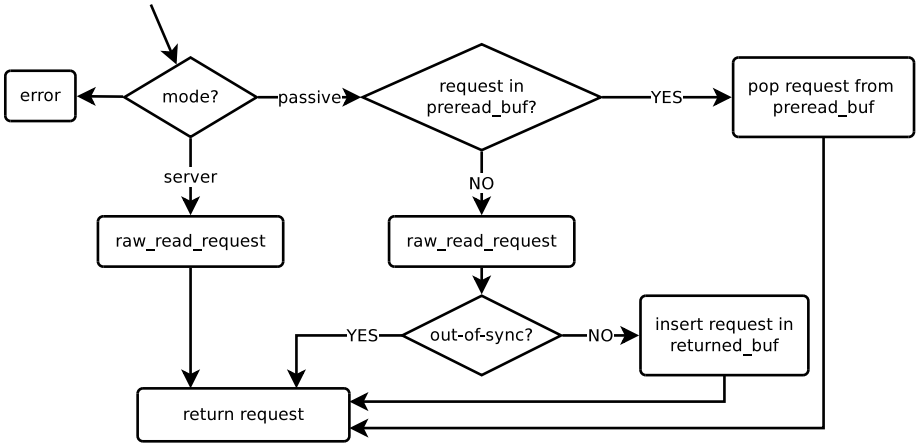
server – v tomto režimu jsou otvírány kanály reprezentující příchozí spojení klienta. Instance vystupuje jako strana serveru a může číst dotazy (*uplink*) a zapisovat odpovědi (*downlink*).

Reference na datové kanály jsou uloženy v atributech `_uplink` a `_dowlink`. Pokud je některý z kanálů nedostupný již při otvírání toku, může být parametr volání `openf()` nahrazen prázdnou hodnout `None`. V takovém případě je tok otevřen jako částečný a je automaticky označen jako *out-of-sync* a čtení zpráv je omezeno (viz odstavec *Ztráta synchronizace toku* 5.5.3).

Čtení a zápis zpráv

Čtení a zápis zpráv je prováděno prostřednictvím metod `read_request()`, `read_response()` a `write_request()`, `write_response()`. Jediným parametrem zápisových metod je zapisovaná zpráva. Vnitřní implementace čtení a zápisu zpráv přistupuje k datovým proudům `ByteStream` přes blokující metody, proto je i čtení a zápis zpráv blokující operací.

Hlavním úkolem třídy `Flow` je zajistit korektní párování dotazů a odpovědí, což představuje první fázi rekonstrukce webového provozu. Správné přečtení odpovědi je navíc podmíněno znalostí metody příslušejícího dotazu, aby mohla být určena dálka jejího těla (viz obrázek 2.7 a jeho popis).



Obrázek 5.3: Znázornění algoritmu metody `read_request()`.

Pro dosažení robustnosti z hlediska různého pořadí volání metod pro čtení a zápis zpráv byly implementovány algoritmy, které využívají následující sdílené fronty pro uchovávání dotazů:

`preread_buf` – ukládá celé instance přednačtených dotazů při opakovanmé čtení pouze odpovědí. Při čtení dotazů jsou nejprve postupně vraceny dotazy z této fronty. Použito v *pasivním* módu.

`returned_buf` – ukládá metody dotazů, které byly opakoveně čteny bez přečtení odpovídajících odpovědí. Pro určení délek následně čtených odpovědí jsou použita data z této fronty. Použito v *pasivním* módu.

`last_written` – zde jsou uloženy metody posledně zapsaných dotazů, aby mohly být určeny délky posléze čtených odpovědí. Tato fronta je použita v režimu *klient*.

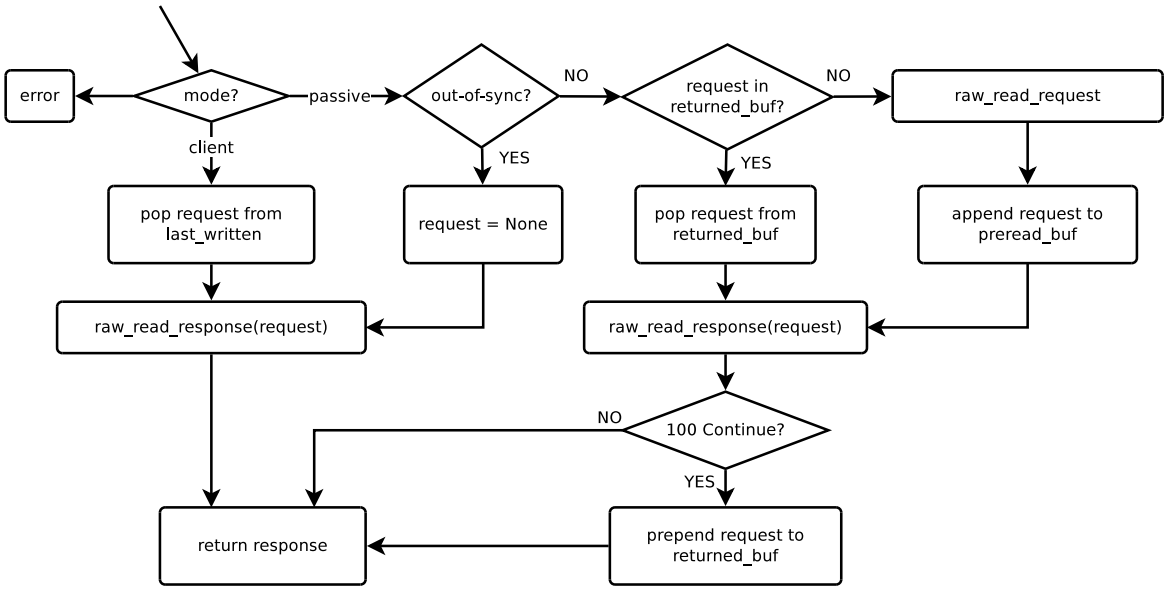
Algoritmy čtení dotazů a odpovědí jsou znázorněny diagramy 5.3 a 5.4. Jejich slovní popis by byl zdlouhavý a nepřehledný. Metody `raw_read_request()` a `raw_read_response()` implementují čtení zpráv přímo z kanálů *uplink* a *downlink*. Jejich úkolem je také odchytávání výjimek vyvolaných při čtení zpráv (metoda `read` třídy `HTTPMessage`) a nastavování příznaku `out-of-sync`.

Z uvedených schémat je patrné, že implementovaná metoda `read_response()` řeší i případy, kdy se v komunikaci objeví vnořená odpověď *100 Continue*. Detailní popis algoritmů lze vyčíst z dokumentace zdrojových kódů projektu.

Ztráta synchronizace toku

Ztráta synchronizace toku znamená, že během čtení některého z kanálů došlo k chybě, což je indikováno vyvoláním příslušné výjimky a nastavením atributu `oos` (*out-of-sync*). Tento stav značí, že není známa přesná pozice „čtecí hlavy“ v rámci datového proudu a tedy není možné korektně určit začátky a konce zpráv.

Zotavení z tohoto stavu je provedeno postupným čtením řádků z chybného kanálu a testováním, zda-li takto přečtený řádek zahajuje HTTP zprávu. Tímto ovšem dojde k postupnému přeskočení zpráv, jejichž prvnímu řádku bezprostředně nepředchází sekvence `CRLF`.



Obrázek 5.4: Znázornění algoritmu metody `read_response()`.

V důsledku tohoto nelze již garantovat korektní párování dotazů a odpovědí a tok je tak mimo synchronizaci (*out-of-sync*).

Čtení odpovědí z desynchronizovaného toku je možné pouze v omezené míře, neboť nejsou známy metody příslušejících dotazů (nelze vytvořit páry, těla odpovědí jsou ignorována). Čtení dotazů je možné bez omezení.

5.5.4 FlowReader

Vyšší abstraktní rohzraní pro práci s HTTP toky. V současné době je třída implementována pouze částečně, a to konkrétně detekce změny spojení na šifrované (hlavička *Upgrade*) a detekce otevření tunelového spojení (metoda `CONNECT`).

Jednotka `FlowReader` není v současné době v nástroji zapojena.

5.5.5 Packet

Pro rekonstrukci a analýzu webového provozu je nezbytná dostupnost informací z hlaviček protokolů síťové vrstvy (IP). Pro tyto účely byla navržena a implementována třída `Packet`, která uchovává podstatné atributy jako zdrojová a cílová IP adresa, číslo zdrojového a cílového portu a časovou značku informující o čase přijetí paketů nesoucích HTTP zprávu.

Nastavování atributů provádějí vstupní adaptéry `CaptureDirectory` a `Proxy` při čtení zpráv ze vstupních toků `Flow`.

Informace poskytované třídou `Packet` mohou být využity například při exportu záznamů o rekonstruci do výstupní XML zprávy. Na obrázku 5.5 je znázorněn digram třídy `Packet` a tříd odvozených.

5.5.6 HTTPMessage

Abstraktní bázová třída pro HTTP zprávy poskyjící rozhraní pro čtení zpráv z datových proudů (metoda `read()`) a jejich zápis do datových proudů (`raw()`). Třída obsahuje

atributy `version` (verze HTTP protokolu), `headers` (hlavičky zprávy) a `entity` (entita zprávy). Popisu hlaviček a entity jsou věnovány samostané podkapitoly dále v textu. Obrázek 5.5 znázorňuje digram třídy a tříd z ní odvozených.

Instance odvozených tříd představují základní datové jednotky, se kterými pracuje modul `cache` a `export`. Zprávy jsou čteny a zapisovány uvnitř toku `Flow` (viz 5.5.3).

Čtení zpráv ze vstupního datového proudu metodou `read()` probíhá v těchto krocích:

1. Zpracování prvního řádku metodou `_firstline()`, kdy se určí metoda, URI a verze protokolu v případě dotazu nebo verze protokolu, stavový kód a důvod v případě. Chyba při zpracování je hlášena výjimkou `BadFirstline`, zpráva nemůže být přečtena.

2. Parsování hlaviček zprávy, třída `HTTPHeaders` a její metoda `read()`. Chyba zpracování je hlášena výjimkou `BadHeaders`, zpráva je poškozená, ale částečně čitelná (minimálně údaje z prvního řádku).

3. Přečtení těla zprávy metodou `_read_body()`, kterou implementují odvozené třídy. Chyba čtení signalizována výjimkou `InvalidBodyRead`, zpráva je částečně čitelná (platné jsou informace z prvního řádku a hlavičky).

V případě chyb čtení zpráv ohlášených uvedenými výjimkami dojde k desynchronizaci toku, ze kterého jsou zprávy čteny.

Zápis zpráv do toku je proráveděn jejich převodem na řetězec bajtů, které jsou pak zapsány do výstupního datového proudu (implementuje třída `Flow`). Převod implementuje metoda `raw()`, která postupně volá metody `raw()` pro první řádek, hlavičky a tělo zprávy a vrátí jejich konkatenované výstupy.

Rozhraní zpráv bylo navrženo a implementováno tak, aby bylo možné jednoduše číst atributy zpráv a také je nastavovat. Toto je realizováno vytvořením atributů jako *property object* (vestavěná funkce `property` a použití dekorátorů, viz [3]), kdy pro každý atribut jsou implementovány metody `getter()`, `setter(value)` a `deleter()`. Při zápisu hodnot přes volání `setter` metody je kontrolována konzistence atributů tak, aby odpovídala specifikaci HTTP/1.1 uvedené v [12, kap. 9, kap. 14].

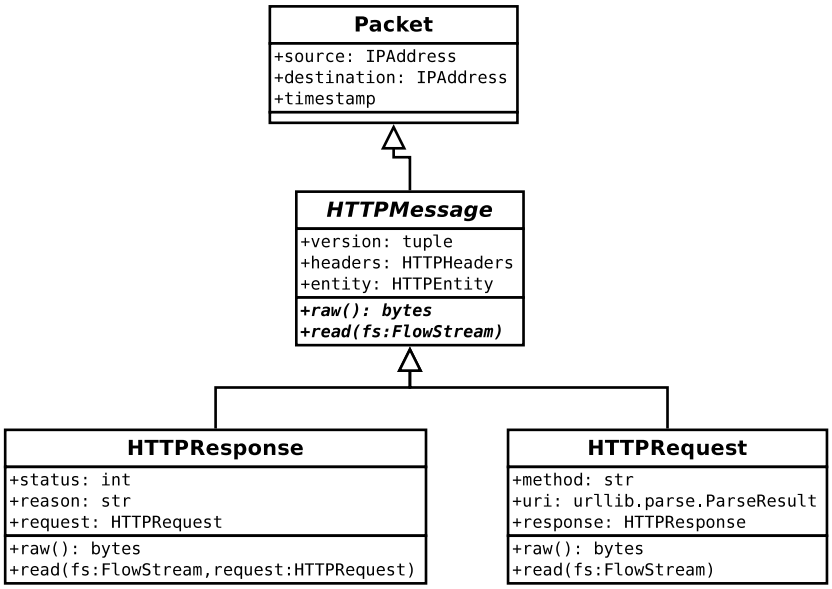
HTTPRequest

Třída reprezentující HTTP dotaz, rozšiřuje rodičovskou třídu o atribut `method` (použitá metoda dotazu) a `request_uri` (adresovaný zdroj), které odpovídají položkám dotazovacího řádku (2.1.3). Přidává také atribut `absolute_uri`, udávající absolutní adresu zdroje určenou podle schéma 2.5.

HTTPResponse

Analogicky pro HTTP odpověď byla implementována třída `HTTPResponse`, jejíž atributy jsou `status` (stavový kód) a `reason` (textový popis stavu).

Metoda `read()` je rozšířena o druhý povinný parametr, který udává metodu příslušejícího dotazu. Na základě metody je možné korektně určit délku těla odpovědi jak bylo popsáno v kapitole 2.1.3



Obrázek 5.5: Diagram třídy `HTTPMessage` a odvozených tříd.

HTTPHeaders

Zapouzdření hlaviček HTTP zprávy, každá zpráva obsahuje instanci této třídy. Pro vnitřní reprezentace hlaviček je použita třída `email.message.Message`, která implementuje formát hlaviček podle [19]. Třída `HTTPHeaders` rozšiřuje rozhraní třídy `Message` (primárně určena emailové zprávy) a adaptuje ji pro použití v HTTP protokolu.

HTTPEntity

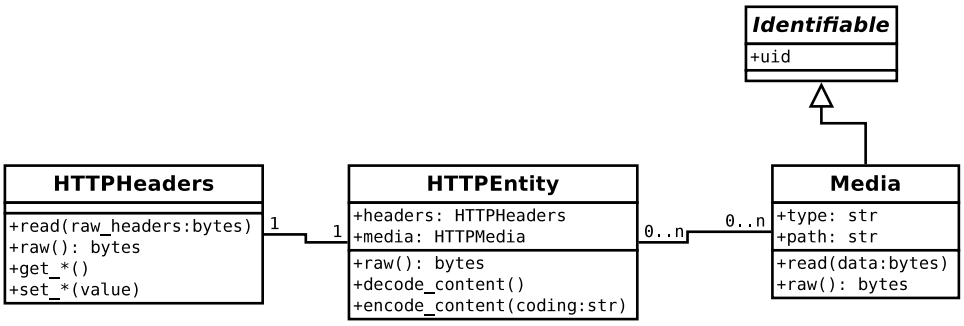
Třída zastupuje HTTP entitu, která sestává z hlaviček a těla. Hlavičky jsou sdílené se zprávou, do které entita náleží, tělo entity pak tvoří tělo této zprávy. Entita je vždy součástí nějaké zprávy.

Hlavním úkolem této třídy je zpřístupnění dat (médií), která jsou přenášena v těle zprávy. Mimo jiné je implementace třídy zodpovědná za (de)kódování entity a nastavování hlaviček, které informují o typu a velikosti přenášených dat. Správa hlaviček je interní záležitostí třídy a při přístupu k mediím přes atribut `media` je zcela transparentní.

Atribut `media` zpřístupňuje obsažená media, která lze jak číst, tak zapisovat. Datový typ medií reprezentuje třída `Media`, která je popsaná v kapitole 5.5.7.

Čtení těla entity probíhá v následujících krocích:

1. Zpracování informací z hlaviček, určení typu a kódování (hlavičky `Content-Encoding` a `Content-Type`).
2. Dekódování těla entity (pokud bylo použito kódování).
3. Vyhledání handleru v databázi mediálních typů (5.5.7) pro daný datový typ.
4. Předání dat handleru.



Obrázek 5.6: Diagram tříd `HTTPHeaders`, `HTTPEntity` a `Media`.

5.5.7 Media

Třída reprezentuje užitečná data přenášená HTTP protokolem. Tato media jsou uložena v těle HTTP enitity, v hlavičkách entity jsou uložena metadata určující typ a velikost mediálních dat.

Rozhraní poskytuje metody `read()` a `raw()` pro čtení a zápis dat (obdobně jako u výše uvedných tříd). Atribut `SUFFIX` obsahuje koncovku jména souboru, která bude použita při pojmenování exportovaných souborů. Exportu dat se podrobněji věnuje kapitola popisující třídu `Export` (5.5.9).

Metoda `write()` parametrizovaná cestou k výstupnímu adresáři provede zápis binárních dat média do souboru pojemenováného podle `uid` a koncovky `SUFFIX`. Volání zápisu je prováděno z modulu `Export`. Po úspěšném zápisu dat, je nastaven atribut `_path` tak, aby udával cestu k souboru se zapsanými daty a binární data jsou uvolněna z paměti. Pozdější přístup k datům přes metodu `raw()` je odbaven čtením dat ze souboru.

Pro každý datový typ (určený hlavičkou *Content-Type*) může být vytvořena třída odvozená od `Media` a implementovat tak zpracování konkrétního typu mediálních dat. `Media` je výchozí třídou pro všechny typy dat.

Databáze mediálních typů

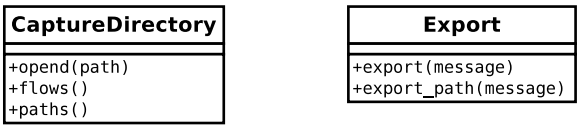
Databáze mediálních typů umožňuje snadno přidávat nové třídy popisující různé datové typy bez nutnosti měnit kód nástroje.

Indexace záznamů databáze je řešena identifikátorem *type/subtupe* (popsán v [12, kap. 3.7]), který rozlišuje datové typy. Modul, který implementuje třídu odvozenou třídu od `Media` pro zpracování určitého datového typu, provede registraci mediální třídy voláním funkce `register_handler(type, handler)`. Tímto se v databázi vytvoří záznam s indexem `type` a s hodnotou `handler`, což je reference na spustitelný objekt (v tomto případě konstruktor mediální třídy).

Entita při čtení hlaviček vyhledá přílušný engine pomocí funkce `get_handler(type)`, aby mu mohla předat tělo entity. Pokud záznam pro index `type` v databázi není, je vrácena výchozí třída `Media`.

5.5.8 CaptureDirectory

Vstupním adaptérem pro statická data (zachycený síťový provoz) je třída `CaptureDirectory`. Jejím vstupem je cesta k adresáři, který obsahuje výstupní soubory nástroje `tcpflow`. Vý-



Obrázek 5.7: Digram tříd `CaptureDirectory` a `Export`

stup třídy je seznam otevřených toků `Flow` pro všechny detekované HTTP konverzace. Diagram třídy je uveden na obrázku 5.7.

Třída implementuje metodu `opend(path)`, která je parametrizovaná textovým řetězcem udávajícím k adresáři se vstupními daty. Vnitřní implementace provede čtení obsahu adresáře, čímž je získán seznam jmen obsažených souborů a adresářů. Ze seznamu jsou vyfiltrována pouze ta jména, která přestavují platné názvy výstupních souborů `tcpflow` ve formátu `XXX.XXX.XXX.XXX:XXXX-YYY.YYY.YYY.YYY:YYYY` ([11]).

Následuje vytvoření dvojic jmen podle zdrojové a cílové IP adresy a čísel portů. Tyto dvojice tvoří jednotlivá zrekonstruovaná TCP spojení. V každé vytvořené dvojici je provedeno rozpoznání kanálů *uplink* a *downlink*, které se provádí opakovaným čtením řádků a testováním, zda-li nejedná o stavový řádek HTTP odpovědi nebo dotazovací řádek HTTP dotazu. Porovnávací metody `is_request(line)` a `is_response(line)` poskytuje balíček `my_http`.

Na závěr je z rozpoznaných kanálů otevřen tok `Flow` v pasivním režimu. Třída poskytuje metody pro sekvenční čtení otevřených toků (`flows()`) nebo jen dvojic jmen souborů (`paths()`).

5.5.9 Export

Modul `Export`, reprezentovaný stejnojmennou třídou, je zodpovědný za exportování mediálního obsahu zpráv do lokálního souborového systému. Diagram této třídy je znázorněn na obrázku 5.7.

Výstupní adresářová struktura tvoří strom, jehož kořenem je adresář dostupný z konfiguračního objektu `CONFIG.EXPORT_DIRECTORY`, parametrem `CONFIG.EXPORT_EN` lze povolit/zakázat export dat. Implementovány jsou dvě varianty budování stromu, první je plochý model *flat* – všechny soubory jsou uloženy v kořenovém adresáři. Druhý model buduje adresářovou strukturu podle doménového jména zdroje z něhož pochází exportovaná data. Strom je v tomto případě budován od první úrovně k nižším a lze omezit jeho maximální výšku parametrem `CONFIG.EXPORT_TREE_MAXDEPTH`. Nastavení modelu stromu se provádí parametrem `CONFIG.EXPORT_TREE_MODE`.

Vstupní metodě `export()` je předána HTTP zpráva, jejíž entita má být exportována. Metoda `export_path()` určí cestu k adresáři, do kterého bude zapsán výstupní soubor. Media z entity jsou poté exportována voláním metod `write(path)` s cestou k výstupnímu adresáři.

5.5.10 Proxy

Brána proxy tvoří síťové rozhraní, přes které se připojuje klient provádějící vizualizaci rekonstruovaného webového provozu. Brána je implementována jako konkurentní TCP server, kdy pro každý příchozí požadavek je vytvořeno samostané vlákno, jenž provede obsloužení požadavku. Další výhodou vlákny odděleného zpracování je omezení dopadu chyb – chyby

vzniklé při obsluze požadavku vedoucí k předčasnému ukončení běhu vlákna ovlivní pouze toto vlákno a brána běžící v samostatném vlákně zůstane nadále dostupná.

Druhé síťové rozhraní brána poskytuje pro vnitřní potřeby nástroje, kdy pomocí metody `connect(address)` jsou navazována spojení s externími zdrojovými servery. Adresa je dvojice *adresa serveru* (doménové jméno nebo IP adresa), *číslo portu*.

Třída implementující proxy bránu je odvozená ze třídy `TCPServer` dostupné v modulu `socketserver` standardní knihovny. Pro každé příchozí spojení je vytvořeno nové vlákno, jehož obsluha je zapouzdřena do tříd `StreamRequestHandler`. Momentálně jsou implementovány dvě obslužné rutiny (digram tříd je uveden na obrázku 5.8):

SimpleHandler – poskytuje funkčnost jednoduché netransparentní proxy brány. Přijaté požadavky předá na původní zdrojové servery a klientovi přepošle jejich odpověď. Při požadavku `CONNECT` vytvoří tunelové spojení s adresovaným serverem. Nepracuje s jednotkou cache.

CacheHandler – je odvozený od předchozí třídy, ale na rozdíl od jednoduché rutiny spolupracuje s jednotkou cache. Příchozí dotazy (mimo `CONNECT`) se delegují na jednotku cache, která vrátí příslušnou odpověď získanou z lokální databáze.

V případě, že odpověď na dotaz není v cache nalezena (výjimka `CacheMiss`), pak je dotaz předán zdrojovému serveru. Odpověď od serveru se vloží do cache a závoreň se přepošle tázajícímu klientovi.

Spuštění proxy brány je implementováno metodou `start()`, která vytvoří nové vlákno v němž spustí TCP server čekající na příchozí spojení. Tato metoda je neblokující a řízení se vrací zpět do hlavního vlákna, které tuto metodu volalo. Adresu, na které bude proxy naslouchat, lze konfigurovat parametry `CONFIG.PROXY_HOST` a `CONFIG.PROXY_PORT`. Povolení spuštění proxy brány je indikováno příznakem `CONFIG.PROXY_EN`. Volba obslužné rutiny je podmíněna povolením použití jednotky cache.

Zastavení vlákna s běžícím TCP serverem proxy brány provádí hlavní vlákno voláním metody `stop()`. Toto volání je blokující a skončí až po uzavření všech spojení TCP serveru a ukončení všech obslužných vláken.

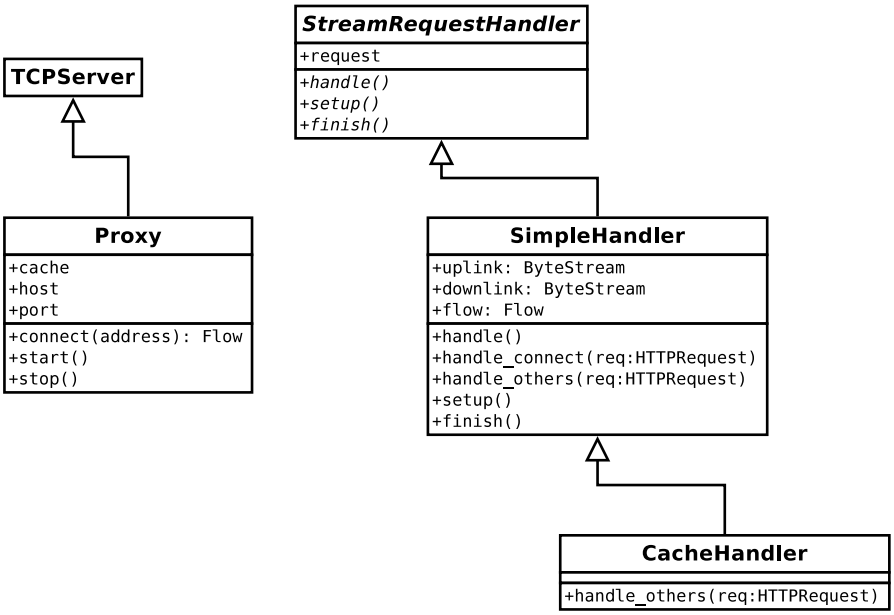
5.5.11 Cache

Jednotka cache tvoří nejdůležitější část rekonstrukčního nástroje. Spravuje databázi záznamů o HTTP komunikaci tak, aby byla schopna odpovídat na dotazy klientů místo původních zdrojových serverů. Klient v ideálním případě nerozezná rozdíl mezi daty vrácenými jedotkou cache a daty, která by poskytnul původní zdrojový server.

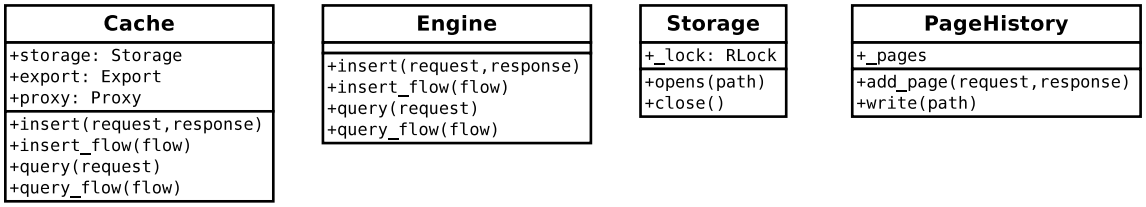
Obecná činnost jednotky cache byla popsána v kapitole 2.3.4, proto se zde zaměříme na řešení její implementace. Třída `Cache` zapouzdřuje celou jednotku cache, vytváří rozhraní pro vkládání a získávání záznamů. Její vnitřní implementace používá pro uložení metadat perzistentní úložiště `Storage`. Celý proces ukládání zpráv do cache a jejich pozdější čtení je řízen řadičem `Engine`.

Propojení s ostatními funkčními bloky nástroje zajišťuje rozhraní sestávající z následujících metod:

`insert(request, response)` – vložení záznamu do cache. Jeden záznam odpovídá jedné dvojici dotaz odpověď. Z dotazu se zjistí URI zdroje, který byl požadován a odpověď pak obsahuje požadovaná data.



Obrázek 5.8: Digram třídy `Proxy`, `SimpleHandler` a `CacheHandler`



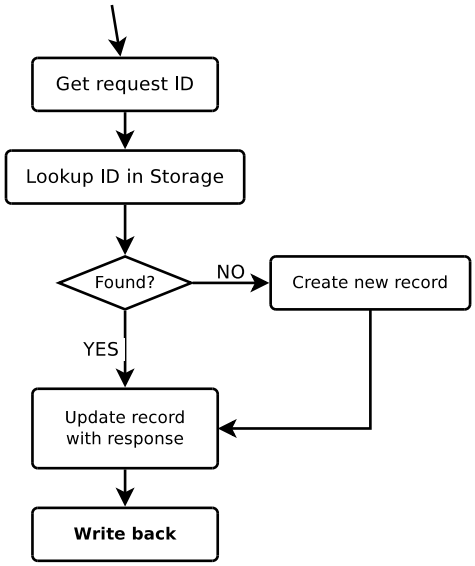
Obrázek 5.9: Diagram tříd `Cache`, `Engine`, `Storage` a `PageHistory`

`insert_flow(flow)` – vložení celého toku. Má stejný význam jako předchozí metoda, akorát vstupem je tok `Flow`. Tato metoda je použitelná pouze pro pasivní toky.

`query(request)` – dotaz na cache pro vyhledání odpovědi. Vrací odpověď na dotaz `request` v případě, že byl nalezen odpovídající záznam. Vyvolá výjimku `CacheMiss` pokud nebyl nalezen záznam, který by odpovídal požadavku dotazu.

`query_flow(flow)` – místo dotazování jednotlivých zpráv je předán ke zpracování rovnou celý tok. V tomto případě cache sama aktivně čte dotazy a odpovídá na ně, metoda je použitelná pro aktivní toky.

Pro každý nově příchozí požadavek je vytvořena nová instance řadiče `Engine`, který provede jeho obsloužení. Vytváření nové instance řadiče je z důvodu zabránění vícenásobného přístupu vláken TCP serveru ke sdíleným propěnným, přesněji přístupu k instanci řadiče. Každý požadavek tak je zpracován vlastní instancí řadiče a tím je zajištěna konzistence dat v rámci řadiče.



Obrázek 5.10: Vložení záznamu do cache.

Engine

Třída zapouzdřující řadič jednotky cache, která implementuje algoritmy pro správu cache. Ke své činnosti využívá perzistentní úložiště `Storage`, kde jsou uložena metadata nezbytná pro činnost řadiče. Změna chování jednotky cache se provádí volbou řadiče prostřednictvím konfigurovatelného parametru `CONFIG.CACHE_ENGINE`. Diagram třídy je uveden na obrázku 5.9.

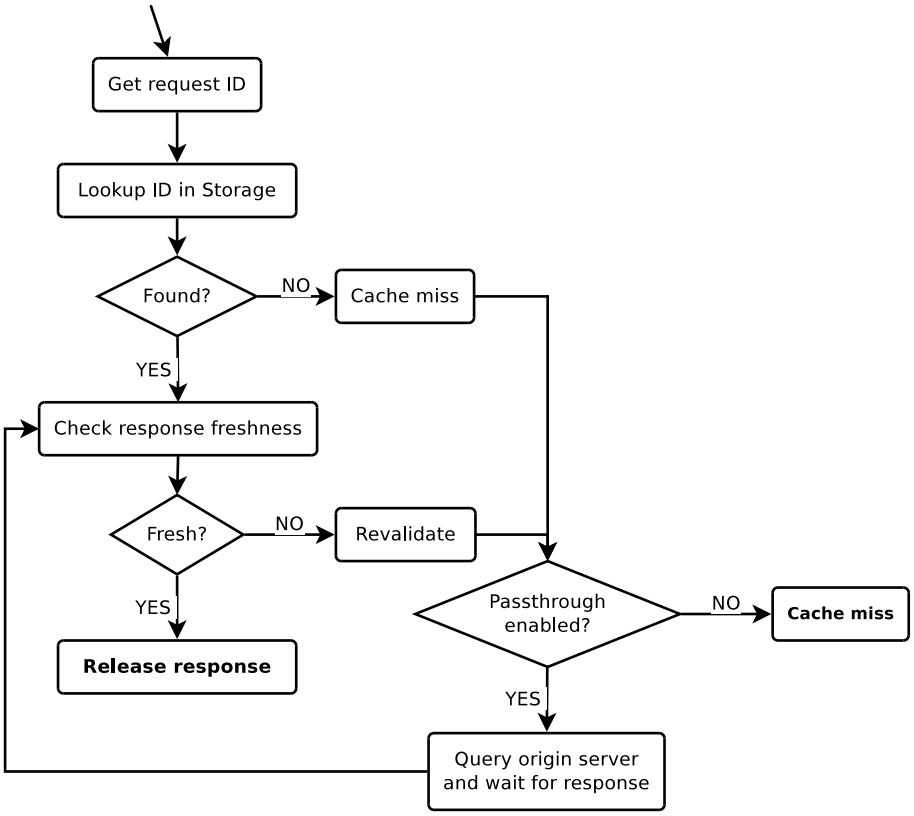
Rozhraní řadiče tvoří metody stejné jaké byly popsány výše pro třídu `Cache`. Volání metod `insert(request, response)` a `query(request)` je delegováno na stejnojmenná volání řadičů. Každá instanci řadiče má dostupné reference na sdílené úložiště `Storage`, exportní modul `Export` a bránu `Proxy`. Řadič tedy může přistupovat do úložiště, provádět export zpráv a navazovat spojení s externími zdrojovými servery.

Obecné algoritmy správy cache implementované řadiči jsou znázorněny diagramy 5.11 a 5.10. Konkrétní implementace řadičů mohou rozšířit tyto základní algoritmy o další výpočetní kroky nebo je naopak zjednodušit. Řadiče implementované v rámci této práce jsou popsány následujícím výčtem.

Engine – výchozí prázdný řadič, který neukládá žádné záznamy a všechny dotazy jsou tak vyhodnoceny jako `CacheMiss`. Jeho cílem je pouze předávat všechny vkládané zprávy do exportního modulu.

RelaxedEngine – „uvolněné řízení“, které povoluje ukládání všech odpovědí bez omezení. Jako identifkátor záznamu je použita absolutní cesta požadovaného zdroje získaná z dotazu. Do úložiště jsou ukládány celé odpovědi včetně hlaviček. Mediální data jsou před vložením odpovědi do úložiště nejprve exportována, aby se ukládala pouze metadata a minimalizovala se tak paměťová složitost záznamů.

Každá vkládaná dvojice dotaz–odpověď je testována, zda-li se nejdená o přenos HTML dokumentu reprezentujícího stránku zobrazitelnou ve webovém prohlížeči (testování provádí mediální třída `Html` jednoduchou analýzou HTML kódu). Pokud je označena jako stránka, pak je vytvořen záznam v historii stránek `PageHistory`.



Obrázek 5.11: Vyhledání záznamu v cache.

Řadič neprovádí testování *čerstvosti* odpovědi, tomu odpovídá přeskočení činnosti *Check response freshness* na digramu 5.11, kdy je každá nalezená odpověď prohlášena za čerstvou a je vrácena jako výsledek vyhledání.

RfcStrictEngine – implementován pouze částečně, má realizovat algoritmus řízení cache podle pravidel uvedných v [12, kap. 13]. Tento řadič interpretuje HTTP hlavičky pro řízení mechanismu cache jako `Cache-Control`, `Date`, `If-Modified-Since` a další.

Tento řadič provádí testování odpovědi na její *čerstvost* a v případě, že je odpověď zastaralá musí nejprve provést její revalidaci dotazem k původnímu zdrojovému serveru.

Do úložiště jsou opět ukládány celé odpovědi až po exportu dat.

Storage

Realizace perzistentního úložiště pro jednotku cache. Úložiště je sdíleno všemi řadiči, aby byla zajištěna konzistence dat při souběžném přístupu bylo implementováno zamykání této datové struktury. Obrázek 5.9 znázorněn diagram třídy.

Otevření databáze provádí metoda `opens(path)`, které je předána cesta k souboru s úložištěm. Není-li provedeno otevření ze souboru, pak je úložiště pouze emulováno v paměti. Zavření úložiště provádí metoda `close()`.

Vnitřní implementace databáze je realizována pomocí databázových objektů standardního modulu `shelve`. Funkce `shelve.open(path)` vrací datový objekt podobný typu *dictionary* (asociativní pole). Na rozdíl od klasického slovníku jsou objekty vkládané do databázového objektu serializovány a zapsány do databázového souboru. Kopírování vkládaných objektů, následná serializace a zápis do souboru je řešen interně modulem `shelve`. Při čtení dat z databázového objektu je aplikován opačný postup. Z pohledu rekonstrukčního nástroje je přístup k datům v databázovém objektu zcela transparentní.

Exkluzivní přístup k úložišti zabezpečuje rekurzivní zámek `RLock` ze standardního modulu `threading`. Všechny přístupové metody třídy `Storage` jsou považovány za kritické sekce a jsou vždy střeženy sdílenou instancí zámku (atribut `_lock`).

PageHistory

Jedná se o abstraktní datovou strukturu pro ukládání historie HTML dokumentů, které byly označeny jako stránky zobrazitelné klientským webovým prohlížečem. Hlavním úkolem této třídy je umožnit snadné ukládání záznamů o historii rekonstrukce webového provozu a její pozdější export do různých formátů podle požadavků klientského prohlížeče, který bude provádět vizualizaci zrekonstruovaného webového provozu.

Metoda `add_page(request, response)` slouží pro přidání záznamu do historie stránek. Vnitřní implementace provede zjištění titulku stránky, vytvoření záznamu obsahujícího požadovaná metadata (např. titulek, časová značka a absolutní URI) a ten uloží ve své lokální databázi.

Export historie do souboru provádí metoda `write(path)`, která se parametrizuje cestou k výstupnímu souboru. V současné době je podporovaný formát historie HTML dokument obsahující reference na zrekonstruované stránky. Údaje jsou uloženy ve stromu podle úrovní doménového jména, přičemž v listových uzlech je zachováno chronologické pořadí záznamů tak, jak byly vkládány do historie.

5.6 Vlastnosti implementace

Implementována byla konkurentní proxy brána, jednotka cache a její dva základní řadiče, adaptér na vstupní statická data a exportní modul. Spouštění nástroje je prováděno pomocí hlavního modulu `main.py`, který poskytuje i textovou nápovědu vysvětlující význam a použití parametrů spouštění.

Úzkým hrdlem implementovaného nástroje je sdílené úložiště `Storage`, ke kterému je vyžadován exkluzivní přístup kvůli zachování konzistence dat. Požadavky na čtení a zápis jsou tak serializovány na interním rekurzivním zámku úložiště. Předpokládá se však, že počet současně přistupujících vláken bude v řádu jednotek (maximálně několik současně připojených vizualizující webových prohlížečů), takže serializace přístupů nebude mít přílišný dopad na dobu odezvy proxy brány.

V současné verzi interpretu CPython jazyka Python 3.2 je omezení v podobě *global interpreter lock*, které umožňuje běh pouze jednoho vlákna v čase v rámci jednoho procesu. Paralelismus na úrovni vláken tak neumí využít potenciál vícejádrových procesorů. Možným řešením by byla obsluhova požadavků klientů v samostatných procesech. Tímto by bylo umožněno skutečné paralelní zpracování, ale za cenu výrazně vyšší režie jádra operačního systému na vytváření procesů.

Kapitola 6

Nasazení rekonstrukčního nástroje

Závěrečná kapitola je věnována testování implementovaného nástroje, shodnocení dosažených výsledků rekonstrukce a diskuzi použitelnosti nástroje.

6.1 Automatizované testování implementace

Během návrhu a implementační fáze byly postupně vytvářeny dílčí testovací moduly, které měly ověřit správnost jednotlivých modulů. V každém balíčku tak vznikl testovací modul `test.py`, který obsahuje sadu automatizovaných testů vytvořených pomocí standardního testovacího prostředí jazyka Python `unittest`. Tyto automatizované testy jsou spouštěny z hlavního testovacího modulu `test.py` umístěného v kořenovém adresáři projektu.

Testovány byly detailní vlastnosti tříd (např. čtení HTTP zpráv ze vstupního datového proudu), ale i zapojení jednotlivých funkčních bloků a jejich použití na vyšší úrovni abstrakce (např. komunikace řadiče cache s webovým serverem prostřednictvím HTTP toku vytvořeného bránou proxy).

Výhodou těchto testů je jejich snadné spouštění, automatizovaný průběh a přehledné vyhodnocení výsledků. Díky tomu mohl být vyvíjený nástroj v průběhu implementace iterativně testován, a tak byla většina chyb odhalována a odstraňována postupně. Tyto testy ovšem nemohly určit kvalitu implementace z pohledu kvality rekonstrukce webového provozu, proto bylo také provedeno testování na reálném provozu.

6.2 Testování na reálném provozu

Nástroj je testován jako celek a cílem testů je vyhodnotit kvalitu rekonstrukce webového provozu. Získání množiny testovacích dat bylo provedeno nástrojem *tcpdump* a zachytávaný webový provoz byl iniciován prohlížečem Chromium 18.0.1025.168 na Ubuntu 11.10. Množina webových serverů, které byly navštíveny, obsahovala webmailové portály (např.: `seznam.cz`, `gmail.com`), statické webové prezentace (např.: `www.fit.vutbr.cz`, `www.vutbr.cz`, `www.mvcr.cz`) a sociální síť (`www.facebook.com`). Před zahájením přístupu na výše uvedené servery byla vymazána interní cache prohlížeče, aby byla všechna data znovu stažena z původních serverů.

Získaný soubor PCAP byl předzpracován programem *tcpflow*, který provedl rekonstrukci TCP toků. Následovalo spuštění rekonstrukčního nástroje, kterému byla jako vstup předána cesta k adresáři s výstupními soubory *tcpflow* a byl spuštěn v režimu proxy brány

(pouze lokální brána bez přístupu na Internet). Výstupem zpracování byl rovněž soubor s hitorií navštívených stránek `history.html`.

K vizualizaci rekonstruovaného webového provozu byl použit prohlížeč Mozilla Firefox 9.0.1, kterému bylo explicitně nastaveno použití proxy brány jako lokálně spuštěné brány rekonstrukčního nástroje. Rovněž byla před zahájením testování vymazána veškerá data z lokální výrovnávací paměti prohlížeče.

6.2.1 Statistická analýza

Problémem hodnocení výsledků rekonstrukce bylo stanovení hodnotících kritérií a získání referenčních dat pro testovací vzorek dat. Nakonec byla zvolena jednoduchá metoda srovnávání počtu a typu zrekonstruovaných HTTP zpráv. Jako referenční výsledky byly vzaty statistiky HTTP zpráv získané z analyzátoru Wireshark. Do nástroje byly přidány potřebné čítače zaznamenávající počty odpovědí a jednotlivých typů dotazů.

Pro získání správného počtu zpráv ve Wiresharku bylo nutné aplikovat filtrační pravidlo pro odstranění duplikovaných zpráv (opakované znovuzaslání ztracených TCP segmentů). Vstupní soubory pro rekonstrukční nástroj tyto duplikovaná data neobsahují, neboť byla provedena rekonstrukce TCP toků pomocí *tcpflow*.

Výsledné počty zpráv ze statistik analyzátoru Wireshark byly shodné s počty zrekonstruovaných zpráv získaných z čítačů rekonstrukčního nástroje. Tento jednoduchý test prokazuje, že rekonstrukční nástroj zpracoval všechny HTTP zprávy ze vstupních souborů.

6.2.2 Výkonnostní analýza

Implementovaný nástroj byl podroben výkonnostní analýze, která měla zjistit časovou a prostorovou složitost webové rekonstrukce a také určit kritická místa kódu z pohledu výkonu. Pro tyto účely byl vytvořen speciální modul `profile.py`, který obsahuje sadu testů na získání potřebných statistických dat. Časové profilování je řešeno standardním modulem `cProfile`, využití paměti pak pomocí nástroje *valgrind*.

Měření výkonu probíhalo na testovacím vzorku reálných dat, která jsou dostupná na přiloženém CD. Nástroj byl postupně spouštěn s různými konfiguracemi a výsledky testů byly zaznamenávány do tabulek pro pozdější analýzu.

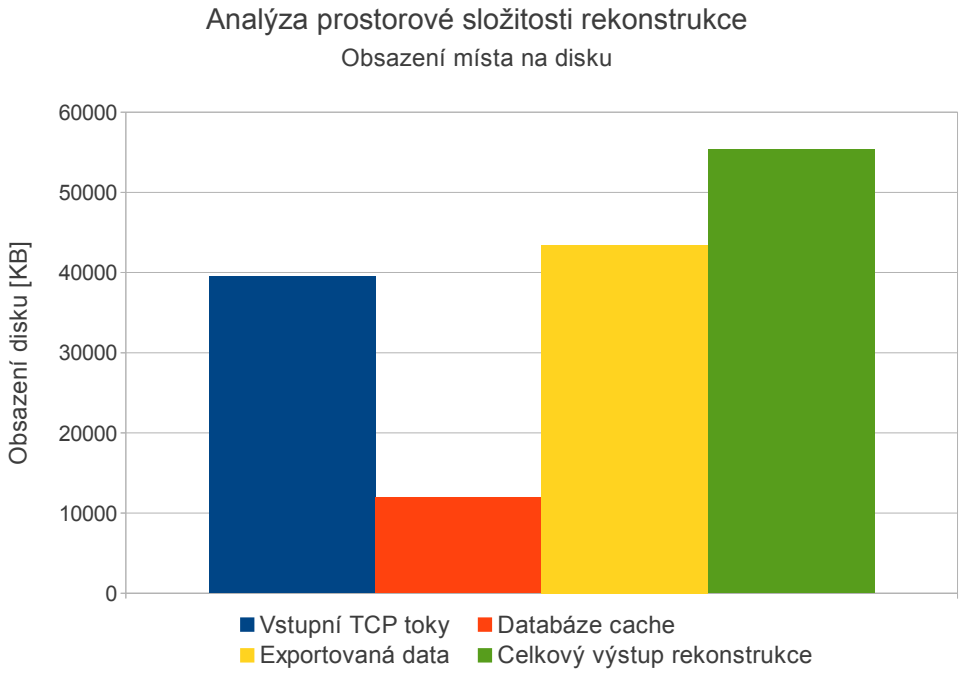
Prostorová složitost

Cílem je určit prostorou složitost rekonstrukce webového provozu, a to jak paměťovou, tak využití pevného disku. Graf 6.1 zobrazuje výsledky běhu v konfiguraci: povolen export dat a databáze cache je uložena na disku.

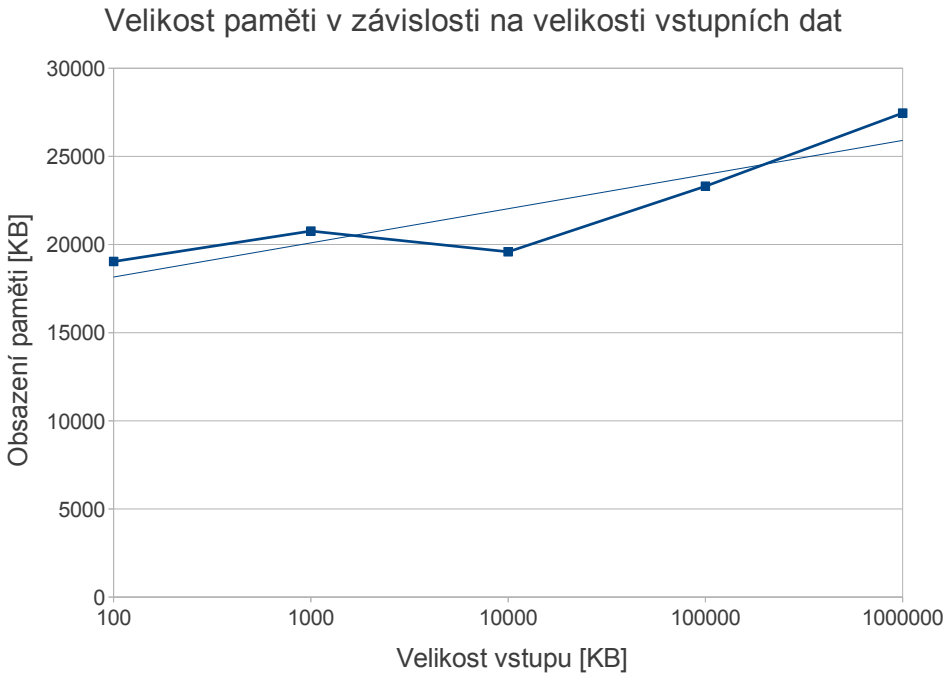
Modrý sloupec vyjadřuje kumulativní velikost vstupních TCP toků v MB (výstupy z *tcpflow*) a zelený pak celkový výstup rekonstrukce. Zbývající dva sloupce vyjadřují rozdělení obsazeného místa na diku mezi databázi cache a exportovaná data.

Graf 6.2 vyjadřuje velikost alokované paměti pro běh nástroje v závislosti na velikosti vstupních dat. Měření bylo provedeno pomocí nástroje *valgrind*, a to konkrétně modulem *Massif*. Měřena byla maximální velikost alokované paměti (sdílená hromada + zásobníky funkcí).

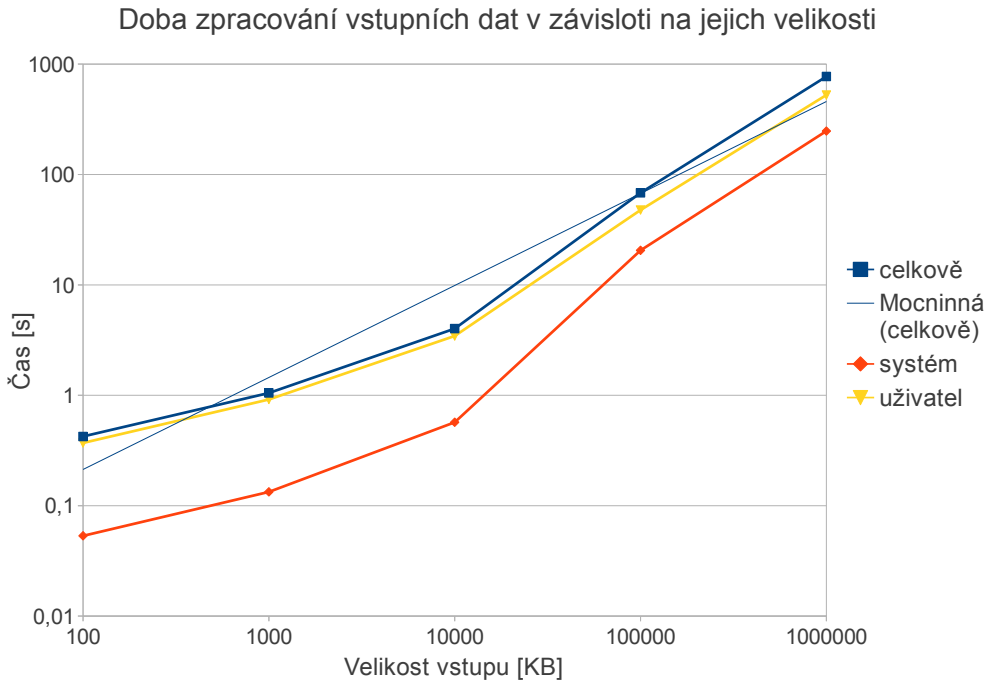
Z uvedného grafu je patrné, že paměťová složitost se blíží logaritmické třídě složitosti. Pro názornost byly naměřené hodnoty proloženy regresní logaritmickou křivkou.



Obrázek 6.1: Prostorová analýza – obsazení disku



Obrázek 6.2: Prostorová analýza – využití paměti



Obrázek 6.3: Časová analýza

Časová složitost

Měření časové složitosti probíhalo pomocí nástroje *time*. Cílem bylo zjistit dobu trvání rekonstrukce vstupních dat v závislosti na jejich velikosti. Aby se minimalizoval dopad přepínání mezi běžícími procesy a vliv různých délek systémových volání, byla provedena opakovaná měření. Získané hodnoty byly zprůměrovány a zanesy do grafu 6.3.

Graf znázorňuje hodnoty zvlášť pro běh v systémovém režimu a v uživatelském režimu. Je zde také doplněna křivka znázorňující celkový čas (součet předchozích), která je proložena mocninnou regresní křivkou. Z tohoto lze usuzovat, že časová složitost rekonstrukce patří do polynomiální třídy.

6.2.3 Výsledky rekonstrukce

Druhým typem analýzy výsledků rekonstrukce webového provozu bylo vizuální porovnání původních dokumentů otevřených v oknech prohlížeče *Chromium* s rekonstruovanými dokumenty zobrazenými prohlížečem *Firefox*. Nutno doplnit, že rekonstrukční nástroj byl nakonfigurován pro použití uvolněného (*relaxed*) řadiče jednotky cache a byl povolen export mediálních dat.

Dosažené výsledky pro statické webové prezentace odpovídaly přesně původně zobrazeným dokumentům až na detailní odlišnosti způsobené rozdílnými renderovacími jádry obou prohlížečů. Tento výsledek byl předpokládaný, neboť v zachyceném síťovém provozu byly přeneseny všechny prvky HTML dokumentů (vyrovnávací paměti prohlížečů byly vymazány) a do jednotky cache byly vloženy všechny HTTP zprávy.

V případě webmailových služeb využívajících šifrované spojení nebyla rekonstrukce možná (zobrazit šlo pouze přihlašovací formuláře), neboť nástroj není schopen dešifrovat

SSL spojení. Překvapivých výsledků bylo dosaženo u dynamických stránek sociální sítě, kdy rekonstruované stránky téměř odpovídalý původním zobrazeným až na části jež využívaly asynchronních komunikačních kanálů.

Výsledkem testu tedy je, že rekonstrukční nástroj korektně uložil zpracované zprávy do vyrovnávací paměti cache. Řadič cache pak správně vybavil uložené odpovědi na dotazy webového prohlížeče provádějícího vizualizaci rekonstruovaného provozu.

Kapitola 7

Závěr

Cílem diplomové práce bylo nastudování problematiky analýzy a rekonstrukce webového provozu, implementování rekonstrukčního nástroje a ověření jeho vlastností. Uvedné cíle byly splněny.

Teoretická část v kapitole 2 seznámila čtenáře s protokolem HTTP a načrtla možné přístupy k řešení rekonstrukce webového provozu. První variantou byla syntaktická analýza HTML kódu s přepisováním referencí, jejíž výhodou byla jednoduchá implementace díky použití analyzátoru Wireshark. Druhý přístup vychází z myšlenky použití HTTP proxy brány s vyrovnávací pamětí cache. Informace nezbytné pro sepsání této kapitoly byly čerpány převážně z RFC dokumentů jež specifikují použité protokoly.

Prvním výstupem práce je implementace architektury využívající syntaktické analýzy (kap. 3). Následné testování ovšem prokázalo omezení vycházejí z koncepce tohoto řešení, proto bylo od architektury upuštěno. Návrh a implementace ovšem přinesly cenné zkušenosti, které nalezly uplatnění při budování nové architektury.

Hlavním očekávaným přínosem přechodu k metodě založené na použití HTTP proxy cache bylo dosažení větší univerzálnosti nástroje. Nová architektura již neprovádí interpretaci přenášených aplikačních dat. Ta je ponechána na vizualizéru, kterým je běžný webový prohlížeč. Rekonstrukční nástroj je tak plně zaměřen pouze na protokol HTTP. Kapitola 4 popisuje návrh nástroje podle této metody.

Nejrozsáhlejší kapitola 5 se snaží poskytnout komplexní pohled na implementaci nástroje. V jejím úvodu je obhájena volba implementačního prostředí následovaná popis struktury projektu (balíčky). Hlavní stať pak tvoří podkapitoly věnované implementaci význačných modulů a tříd. Závěr kapitoly diskutuje vlastnosti implementace.

Samostaná kapitola 6 je věnována testování analýze výsledného nástroje. Je zde představen systém automatizovaných testů a také výkonnostních testů na reálných datech. Jsou nastavena hodnotící kriteria a popsány způsoby získávání referenčních výstupů. Závěr kapitoly přehledně uvádí dosažené výsledky a komentuje jejich význam.

Hlavním přínosem dimplomové práce je navržení a implementování rekonstrukčního nástroje, který tvoří funkční základnu (resp. knihovnu pro práci s protokolem HTTP) využitelnou pro rozšiřující moduly implementující pokročilé metody analýzy konktrétních aplikací vystavěných nad protokolem HTTP.

Vhodným pokračováním práce by byla právě analýza aplikací vystavěných nad protokolem HTTP. Tato by měla vyústit v implemntaci pokročilých algoritmů řízení jednotky cache, tak aby bylo dosaženo lepších výsledků rekonstrukce. Zajímavou výzvou je pak také řešní posuvu času v rámci cache, kdy si vizualizační nástroj bude schopen explicitně vyžádat data z cache podle určitého data vložení. Čistě praktickým cílem z pohledu reálného

nasazení rekonstrukčního nástroje je optimalizace současného řešení tak, aby poskytovalo dostatečný výkon pro práci s rozsáhlejšími vstupními daty.

Literatura

[1] libxml2dom [stránky projektu].
http://www.boddie.org.uk/python/libxml2dom.html, [cit. 2011-12-20].

[2] PHP: Hypertext Preprocessor [stránky projektu]. http://php.net, [cit. 2012-01-02].

[3] Python Documentation. http://www.python.org/doc.

[4] The Apache HTTP Server Project [stránky projektu]. http://httpd.apache.org, [cit. 2012-01-02].

[5] Wireshark network protocol analyzer [stránky projektu].
http://www.wireshark.org, [cit. 2011-12-30].

[6] Moderní prostředky pro boj s kybernetickou kriminalitou na Internetu nové generace.
http://www.fit.vutbr.cz/research/view_project.php?id=517, 2010–2015, [cit. 2011-12-30].

[7] Berners-Lee, T.: Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. RFC 1630 (Informational), Červen 1994.
URL http://www.ietf.org/rfc/rfc1630.txt

[8] Berners-Lee, T.; Fielding, R.; Frystyk, H.: Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), Květen 1996.
URL http://www.ietf.org/rfc/rfc1945.txt

[9] Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), Květen 1996.
URL http://www.ietf.org/rfc/rfc1951.txt

[10] Deutsch, P.: GZIP file format specification version 4.3. RFC 1952 (Informational), Květen 1996.
URL http://www.ietf.org/rfc/rfc1952.txt

[11] Elson, J.: *tcpflow* Manual page.
http://www.circlemud.org/jelson/software/tcpflow/tcpflow.1.html, [cit. 2012-05-15].

[12] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Červen 1999, updated by RFCs 2817, 5785, 6266.
URL http://www.ietf.org/rfc/rfc2616.txt

[13] Freed, N.; Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), Listopad 1996, updated by RFCs 2184, 2231, 5335.
URL http://www.ietf.org/rfc/rfc2045.txt

[14] Khare, R.; Lawrence, S.: Upgrading to TLS Within HTTP/1.1. RFC 2817 (Proposed Standard), Květen 2000.
URL http://www.ietf.org/rfc/rfc2817.txt

[15] Kristol, D.; Montulli, L.: HTTP State Management Mechanism. RFC 2965 (Historic), Říjen 2000, obsoleted by RFC 6265.
URL http://www.ietf.org/rfc/rfc2965.txt

[16] Lamping, U.: Wireshark Developmer's Guide [programátorská dokumentace]. http://www.wireshark.org/docs/wsdg_html_chunked/, 2004–2010, [cit. 2012-01-07].

[17] Leach, P.; Mealling, M.; Salz, R.: A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), Červenec 2005.
URL http://www.ietf.org/rfc/rfc4122.txt

[18] Protivínský, M.: Internetová kriminalita. *Kriminalistika, čtvrtletník pro kriminalistickou teorii a praxi*, červen 2008, dostupné elektronicky http://aplikace.mvcr.cz/archiv2008/casopisy/kriminalistika/2002/02_02/protivin.html [cit. 2012-01-08].

[19] Resnick, P.: Internet Message Format. RFC 2822 (Proposed Standard), Duben 2001, obsoleted by RFC 5322, updated by RFCs 5335, 5336.
URL http://www.ietf.org/rfc/rfc2822.txt

Příloha A

Obsah CD

Přiložené CD obsahuje zdrojové kódy rekonstrukčního nástroje využívajícího paketový analyzátor Wireshark, zdrojové kódy rekonstrukčního nástroje podle nové arcitektury. Dále jsou obsaženy zdrojové kódy této technické zprávy ve formátu LaTeX a také přeložená verze ve formátu PDF.